aws

最佳實務指南

Amazon EKS

Copyright © 2026 2024. Review license at https://github.com/aws/aws-eks-best-practices/blob/master/LICENSE

Amazon EKS: 最佳實務指南

Copyright © 2026 2024. Review license at https://github.com/aws/aws-eks-best-practices/blob/master/LICENSE

Amazon 的商標和商業外觀不得用於任何非 Amazon 的產品或服務，也不能以任何可能造成客戶混淆、任何貶低或使 Amazon 名譽受損的方式使用 Amazon 的商標和商業外觀。所有其他非 Amazon 擁有的商標都是其各自擁有者的資產，這些擁有者不一定與 Amazon 相關聯、連線或贊助

Table of Contents

簡介 1
相關指南 1
做出貢獻 2
安全 3
本指南的使用方式 3
了解共同的責任模型 3
簡介 1
意見回饋 5
深入閱讀 5
工具和資源 6
EKS Auto 模式 - 安全性 6
安全架構 7
常見問答集 (FAQ) 15
身分和存取權管理 18
控制對 EKS 叢集的存取 18
叢集存取建議 24
EKS Pod 的身分和登入資料 30
EKS Pod 建議的身分和登入資料 37
工具與資源 45
Pod 安全性 45
Linux 功能 45
Pod 安全解決方案 47
建議 54
工具和資源 45
多租用 60
軟式多租用戶 60
Kubernetes 建構 62
緩解控制項 64
硬式多租戶 70
未來方向 71
多叢集管理工具和資源 71
Detective 控制項 71
建議 54
工具和資源 45

網路安全 79
流量控制 79
網路加密 79
網路政策 80
建議 54
Security groups (安全群組) 84
何時使用網路政策與 Pod 的安全群組？ 86
Service Mesh 政策強制執行或 Kubernetes 網路政策 87
ThirdParty網路政策引擎 88
傳輸中加密 89
工具和資源 45
資料加密和秘密管理 98
靜態加密 98
秘密管理 100
工具和資源 45
執行期安全 103
安全內容和內建的 Kubernetes 控制項 103
建議 54
工具與資源 105
基礎設施安全性 106
建議 54
替代方案 109
工具和資源 45
法規合規 112
向左移 114
工具和資源 45
事件回應和鑑識 115
事件回應計劃範例 115
建議 54
工具和資源 45
映像安全性 119
建議 54
工具和資源 45
多帳戶策略 128
規劃多租戶叢集的多工作負載帳戶策略 128
集中式 EKS 叢集 128

分散式 EKS 叢集 134
集中式與分散式 EKS 叢集 136
叢集存取管理 137
EKS 存取管理選項 138
選項 1：具有叢集存取管理 (CAM) API 的 AWS IAM Identity Center 138
選項 2：映射至 Kubernetes 群組的 AWS IAM 使用者/角色 140
選項 3：OIDC 供應商 140
工作負載的 AWS EKS Pod 身分與 IRSA 141
建議 141
叢集自動擴展 143
EKS 自動模式 143
使用自動模式的原因 144
常見問答集 144
Karpenter 149
建議 54
Karpenter 最佳實務 150
建立 NodePools 153
排程 Pod 157
CoreDNS 建議 159
Karpenter 藍圖 159
其他資源 78
Cluster Autoscaler 160
概觀 160
針對效能和可擴展性進行最佳化 164
最佳化成本和可用性 166
進階使用案例 168
其他參數 170
其他資源 78
參考 70
可靠性 173
本指南的使用方式 3
簡介 1
意見回饋 5
應用程式 175
建議 54
水平 Pod Autoscaler (HPA) 179

垂直 Pod Autoscaler (VPA) 180
更新應用程式 180
運作狀態檢查和自我修復 182
建議 182
處理中斷 184
建議 185
可觀測性 186
建議 187
控制平面 188
EKS 架構 189
建議 190
監控控制平面指標 190
叢集身分驗證 193
許可 Webhook 195
處理叢集升級 196
叢集端點連線能力 196
執行大型叢集 197
其他資源： 197
資料平面 197
建議 54
聯網 203
Kubernetes 網路模型 203
容器聯網界面 (CNI) 204
Amazon Virtual Private Cloud (VPC) CNI 204
子網路計算器 206
VPC 和子網路考量事項 206
概觀 160
建議 54
Amazon VPC CNI 217
概觀 160
建議 54
最佳化 IP 地址使用率 229
最佳化節點層級 IP 消耗 229
減少 IP 耗盡 229
執行 IPv6 叢集 237
概觀 160

建議 54
自訂聯網 253
範例組態 254
建議 54
Linux 的字首模式 260
建議 54
Windows 的字首模式 265
建議 54
每個 Pod 的安全群組 270
建議 54
負載平衡 278
選擇Load Balancer類型 278
佈建負載平衡器 279
選擇Load Balancer目標類型 280
設定Load Balancer運作狀態檢查 282
可用性和 Pod 生命週期 283
參考 70
附錄 287
監控網路效能問題 288
監控 DNS 限流問題的 CoreDNS 流量 288
使用 Conntrack 指標監控 DNS 查詢延遲 288
其他重要的網路效能指標 289
擷取指標以監控工作負載的網路效能問題 289
在 IPVS 模式下執行 kube-proxy 297
概觀 160
可擴展性 301
本指南的使用方式 301
了解擴展維度 301
超大型擴展 302
控制平面 302
限制工作負載和節點爆量 303
安全地擴展節點和 Pod 303
執行 Kubectl 時使用用戶端快取 304
停用 kubectl 壓縮 304
碎片叢集自動擴展器 305
API 優先順序和公平性 306

擷取 API 伺服器中的資源 312
資料平面 318
自動節點自動擴展 318
使用許多不同的 EC2 執行個體類型 318
偏好較大的節點以減少 API 伺服器負載 319
使用類似的節點大小來取得一致的工作負載效能 320
有效率地使用運算資源 320
自動化 Amazon Machine Image (AMI) 更新 321
針對容器使用多個 EBS 磁碟區 321
如果工作負載使用 EBS 磁碟區，請避免具有低 EBS 連接限制的執行個體 323
停用對磁碟不必要的記錄 323
當作業系統更新速度為必要時，就有修補程式執行個體 323
叢集服務 324
擴展 CoreDNS 324
垂直擴展 Kubernetes 指標伺服器 325
CoreDNS lameduck 持續時間 326
CoreDNS 準備程度探查 326
記錄和監控代理程式 326
工作負載 327
使用 IPv6 進行 Pod 聯網 328
限制每個命名空間的服務數量 328
了解 Elastic Load Balancer 配額 328
使用 Route 53、Global Accelerator 或 CloudFront 329
使用 EndpointSlices 而非 Endpoints 329
如果可能，請使用不可變和外部秘密 329
限制部署歷史記錄 330
預設停用 enableServiceLinks 330
限制每個資源的動態許可 Webhook 331
比較跨多個叢集的工作負載 331
擴展背後的理論 332
節點與流失率 332
每秒查詢數的思考 332
擴展分散式元件 333
上游和下游瓶頸 333
依指標擴展 335
控制平面監控 337

API 伺服器 337
問題在哪裡？ 337
排程器 342
Kube 控制器管理員 344
ETCD 346
節點效率和擴展 347
節點選擇 347
節點儲存貯體封裝 348
使用率與飽和度 355
設定 CPU 限制 362
Kubernetes SLOs 365
Kubernetes SLOs 365
Kubernetes SLI 指標 367
叢集上的 SLOs 370
已知限制Service Quotas 371
其他 AWS Service Quotas 372
AWS 請求調節 376
其他已知限制 377
叢集升級 378
概觀 378
升級之前 378
讓您的叢集保持在up-to-date 379
檢閱 EKS 發行行事曆 379
了解共同責任模型如何套用至叢集升級 380
就地升級叢集 380
依序升級您的控制平面和資料平面 380
使用 EKS 文件建立升級檢查清單 381
使用 Kubernetes API 升級附加元件和元件 381
升級前驗證基本 EKS 需求 382
驗證可用的 IP 地址 383
驗證 EKS IAM 角色 383
遷移至 EKS 附加元件 384
升級控制平面之前，識別並修復已移除的 API 用量 385
叢集洞見 385
Kube-no-trouble 387
Pluto 387

Resources 388
更新 Kubernetes 工作負載。使用 kubectl-convert 更新資訊清單 389
設定 PodDisruptionBudgets 和topologySpreadConstraints，以確保升級資料平面時工作負載的可用性 389
使用受管節點群組或 Karpenter 簡化資料平面升級 391
確認與現有節點和控制平面的版本相容性 391
啟用 Karpenter 受管節點的節點過期 392
針對 Karpenter 受管節點使用漂移功能 392
使用 eksctl 自動化自我管理節點群組的升級 392
升級之前備份叢集 393
升級控制平面後重新啟動 Fargate 部署 393
評估藍/綠叢集作為就地叢集升級的替代方案 393
追蹤 Kubernetes 專案中計劃的主要變更 — 預先考慮 394
功能移除的特定指引 394
移除 1.25 中的 Dockershim - 使用 Detector for Docker Socket (DDS) 394
移除 1.25 中的 PodSecurityPolicy - 遷移至 Pod 安全標準或policy-as-code解決方案 394
在 1.23 中棄用樹狀儲存驅動程式 - 遷移至容器儲存介面 (CSI) 驅動程式 395
其他資源 395
ClowdHaus EKS 升級指南 395
GoNoGo 395
成本最佳化 396
一般準則 396
EKS 成本最佳化最佳實務 396
本指南的使用方式 3
AWS 服務和 Kubernetes 的主要功能 397
意見回饋 5
架構 397
請參閱支柱：衡量和責任 398
儲存支柱：成本最佳化 398
計畫支柱：規劃和預測 399
執行支柱 399
參考 70
意識 400
建議 54
其他工具 407
運算 408

適當調整工作負載大小 408
減少耗用量 409
減少未使用的容量 410
最佳化運算容量類型 414
最佳化運算用量 417
網路 418
Pod 到 Pod 通訊 419
Load Balancer與 Pod 通訊 430
從容器登錄檔傳輸資料 432
資料傳輸至網際網路和 AWS 服務 433
VPCs之間的資料傳輸 435
使用服務網格 437
其他資源 78
儲存 445
概觀 160
暫時性容積 445
持久性磁碟區 446
其他考量 451
可觀測性 452
簡介 452
日誌 452
EKS 控制平面 452
EKS 資料平面 453
指標 457
追蹤 459
其他資源： 78
Windows 462
AMI 管理 462
管理您自己的 Amazon EKS 最佳化 Windows AMI 463
為自訂 EKS 最佳化 AMIs 設定更快速的啟動 464
在自訂 AMIs上快取 Windows 基本層 464
部落格文章 466
gMSA for Windows Containers 466
什麼是 gMSA 帳戶 466
Windows 容器和 gMSA 使用案例 466
Windows Server 強化 468

使用 Windows Server Core 減少攻擊面 469
避免 RDP 連線 469
Amazon Inspector 470
Amazon GuardDuty 471
Amazon EC2 for Windows 中的安全性 471
掃描 Windows 映像 471
Windows 版本和授權 472
Windows Server 版本 472
授權 472
記錄 472
記錄建議 473
監控 Windows 容器 474
Windows 網路 478
Windows 容器聯網概觀 478
IP 地址管理 479
容器網路界面 (CNI) 選項 481
網路政策 482
記憶體和系統管理 482
保留系統和 kubelet 記憶體 482
Windows 容器記憶體需求 483
結論 483
基礎設施管理 483
推送和提取 Windows 映像 484
參考資料 486
排程 486
將 PODs 指派給節點最佳實務 486
確保作業系統特定的工作負載登陸適當的容器主機 486
處理相同叢集中的多個 Windows 組建 487
使用 RuntimeClass 簡化 Pod 資訊清單中的 NodeSelector 和公差 489
受管節點群組支援 490
其他文件 490
Windows 容器的 Pod 安全性 490
儲存選項 493
什麼是樹狀內與out-of-tree磁碟區外掛程式？ 493
適用於 Windows 的樹狀內磁碟區外掛程式 493
Out-of-tree 497

Amazon FSx for Windows File Server 497
強化 Windows 容器映像 498
1. 使用本機安全政策和登錄檔設定帳戶政策 （密碼或鎖定） 499
2. 稽核政策 501
3. Windows 容器的 IIS 安全最佳實務 503
4. 最低權限準則 507
最終想法：為什麼保護您的 Windows 容器是現今威脅環境的必要條件 509
混合 511
網路中斷連線 511
最佳實務 511
Kubernetes Pod 容錯移轉 516
應用程式網路流量 521
主機登入資料 525
AI/ML 528
意見回饋 528
運算 528
GPU 資源最佳化和成本管理 528
節點彈性和訓練任務管理 539
應用程式擴展和效能 541
進階 GPU 管理的動態資源配置 543
CPU 推論 587
概觀 160
為什麼選擇適用於 AI 工作負載的 CPUs？ 589
CPU 與 GPU/ Trainium：何時選擇每個 589
工作負載決策架構 590
生產模式 592
最佳化最佳實務 593
評估 CPU 優先工作負載的模型品質 597
參考 70
聯網 599
對於具有高節點間通訊的應用程式，請考慮較高的網路頻寬或彈性布料轉接器 600
在大型 GPU 執行個體上規劃 IP 地址使用 600
安全 601
安全與合規 601
儲存 602
資料管理和儲存 602

可觀測性 612
監控與可觀測性 612
可觀測性和指標 613
效能 626
管理 ML 成品、服務架構和啟動最佳化 626
最佳化容器映像提取時間 628
考慮 NVMe for kubelet 和容器儲存 631
貢獻 632
現有參與者的摘要 632
設定本機編輯環境 632
分叉並複製儲存庫 632
開啟 VS 程式碼工作區 632
編輯檔案 633
提交提取請求 633
使用 github.dev Web 型編輯器 633
編輯單一頁面 634
檢視並設定頁面的 ID 634
設定頁面 ID 634
建立新頁面 635
建立頁面中繼資料 635
新增至目錄 635
插入映像 636
使用 Vale 檢查樣式 636
建置本機預覽 636
AsciiDoc Cheat Sheet 637
基本格式化 637
標頭 637
清單 637
連結 638
映像 638
程式碼區塊 638
表格 638
輔助 639
包括 639
..... dcxl

Amazon EKS 最佳實務指南

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

歡迎使用 EKS 最佳實務指南。此專案的主要目標是為 Amazon EKS 的第 2 天操作提供一組最佳實務。我們選擇將此指引發佈至 GitHub，以便快速迭代、為各種問題提供及時且有效的建議，並輕鬆納入更廣泛的社群的建議。

我們目前已發佈下列主題的指南：

- 安全性的最佳實務
- 可靠性的最佳實務
- 叢集自動擴展的最佳實務：Karpenter
- Cluster Autoscaling 的最佳實務：cluster-autoscaler
- 叢集自動擴展的最佳實務：EKS Auto 模式
- 聯網的最佳實務
- 可擴展性的最佳實務
- 叢集升級的最佳實務
- 成本最佳化的最佳實務
- 執行 Windows 容器的最佳實務
- 混合部署的最佳實務
- 執行 AI/ML 工作負載的最佳實務

我們也開啟了名為 hardeneks 的 Python 型 CLI（命令列界面），以檢查本指南中的一些建議。

未來我們將發佈效能、成本最佳化和卓越營運方面的最佳實務指引。

相關指南

除了 EKS 使用者指南之外，AWS 還發佈了幾個其他指南，可協助您實作 EKS。

- EMR 容器最佳實務指南

- EKS 上的資料
- AWS 可觀測性最佳實務
- Terraform 的 Amazon EKS 藍圖
- Amazon EKS 藍圖快速入門

做出貢獻

我們鼓勵您為這些指南做出貢獻。如果您已實作證實有效的實務，請開啟問題或提取請求來與我們分享。同樣地，如果您在我們已發佈的指引中發現錯誤或瑕疵，請提交 PR 進行更正。您可以在我們的貢獻指導方針中找到提交 PRs的指導方針。 https://github.com/aws/aws-eks-best-practices/blob/master/CONTRIBUTING.md

安全性的最佳實務

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

本指南提供保護依賴 EKS 的資訊、系統和資產的建議，同時透過風險評估和緩解策略提供商業價值。此處的指引是 AWS 發佈的一系列最佳實務指南的一部分，可協助客戶根據最佳實務實作 EKS。績效、卓越營運、成本最佳化和可靠性指南將在未來幾個月提供。

本指南的使用方式

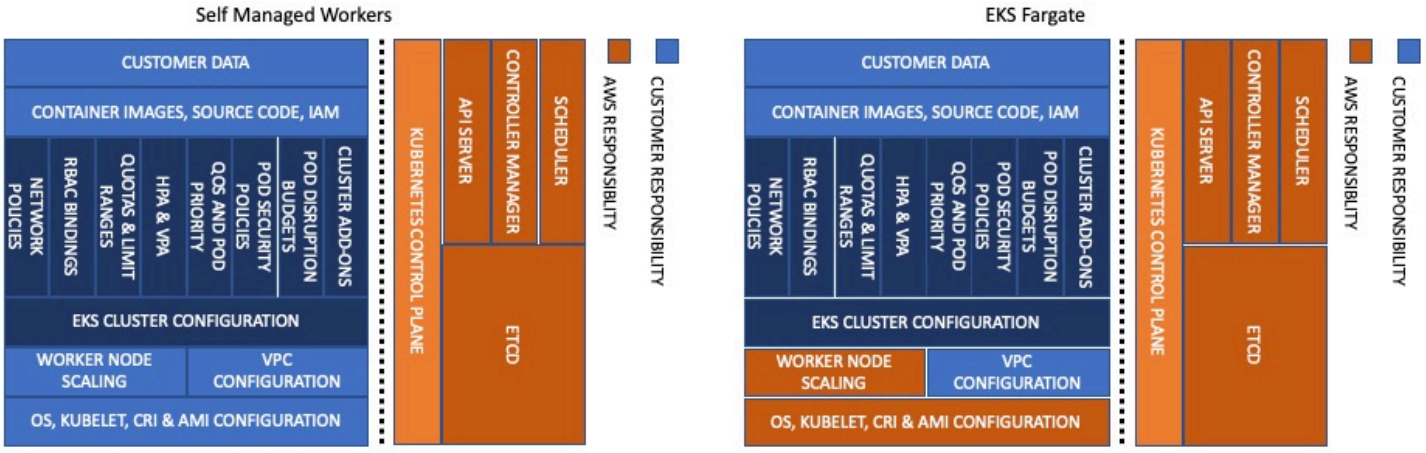
本指南適用於負責實作和監控 EKS 叢集及其支援工作負載之安全控制有效性的安全從業人員。本指南會組織成不同的主題區域，以便於使用。每個主題都從簡短的概觀開始，接著是保護 EKS 叢集的建議和最佳實務清單。主題不需要以特定順序讀取。

了解共同的責任模型

使用 EKS 等受管服務時，安全性和合規性被視為共同的責任。一般而言，AWS 負責「雲端」的安全，而身為客戶的您負責「雲端」的安全。使用 EKS，AWS 負責管理 EKS 受管 Kubernetes 控制平面。這包括 Kubernetes 控制平面節點、ETCD 資料庫，以及 AWS 提供安全可靠服務所需的其他基礎設施。身為 EKS 的消費者，您主要負責本指南中的主題，例如 IAM、Pod 安全性、執行期安全性、網路安全等。

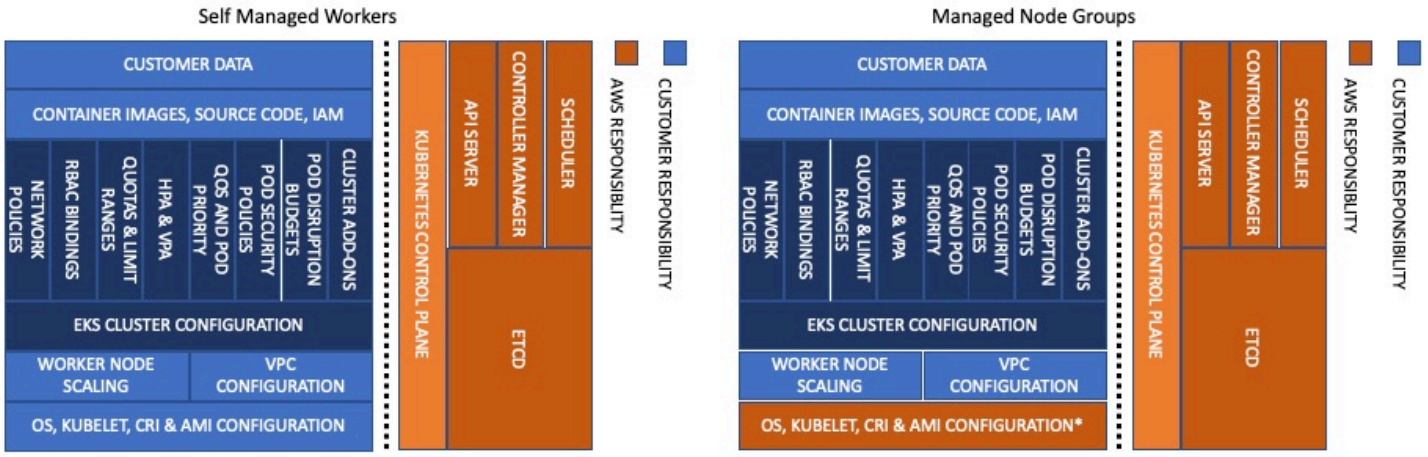
在基礎設施安全方面，當您從自我管理的工作者、受管節點群組到 Fargate 時，AWS 將承擔額外的責任。例如，使用 Fargate，AWS 會負責保護用來執行 Pod 的基礎執行個體/執行時間。

共同責任模型 - Fargate



AWS 也將承擔使用 Kubernetes 修補程式版本和安全性修補程式將 EKS 最佳化 AMI 保持在最新狀態的責任。使用受管節點群組 (MNG) 的客戶負責透過 EKS API、CLI、Cloudformation 或 AWS 主控台將其節點群組升級至最新的 AMI。此外，與 Fargate 不同，MNGs不會自動擴展您的基礎設施/叢集。這可由叢集自動擴展器或其他技術處理，例如 Karpenter、原生 AWS Autoscaling、SpotInst 的 Ocean 或 Atlassian 的 Escalator。

共同責任模型 - MNG



在設計您的系統之前，請務必了解您的責任與服務供應商 (AWS) 之間的分界線。

如需有關共同責任模型的其他資訊，請參閱 https：//https://aws.amazon.com/compliance/shared-responsibility-model/

簡介

使用 EKS 等受管 Kubernetes 服務時，有幾個相關的安全最佳實務領域：

- 身分和存取權管理
- Pod 安全性
- 執行期安全性
- 網路安全
- 多租用
- 多租戶的多帳戶
- Detective 控制項
- 基礎設施安全性
- 資料加密和秘密管理
- 法規合規
- 事件回應和鑑識
- 映像安全性

在設計任何系統時，您需要考慮其安全性影響，以及可能影響安全性狀態的實務。例如，您需要控制誰可以對一組資源執行動作。您也需要能夠快速識別安全事件、保護您的系統和服務免於未經授權的存取，以及透過資料保護來維護資料的機密性和完整性。擁有一組明確定義且經過預演的程序來回應安全事件，也會改善您的安全狀態。這些工具和技術之所以重要，因為能支援諸多目的，例如防止財務損失或遵循法規義務。

AWS 提供一組豐富的安全服務，根據廣泛的安全意識客戶的意見回饋而演進，協助組織實現其安全與合規目標。透過提供高度安全的基礎，客戶可以在「無差別繁重」上花費較少時間，並在實現業務目標上花費更多時間。

意見回饋

本指南已在 GitHub 上發佈，以便收集更廣泛的 EKS/Kubernetes 社群的直接意見回饋和建議。如果您有最佳實務，認為我們應該包含在指南中，請在 GitHub 儲存庫中提出問題或提交 PR。我們的目的是在新功能新增至服務或新的最佳實務演進時，定期更新指南。

深入閱讀

Kubernetes 安全白皮書，由安全稽核工作群組贊助，此白皮書說明 Kubernetes 攻擊面和安全架構的關鍵層面，旨在協助安全從業人員做出健全的設計和實作決策。

CNCF 也發佈了一份有關雲端原生安全性的白皮書。本文探討技術環境如何演進，並倡導採用符合 DevOps 程序和敏捷方法的安全實務。

工具和資源

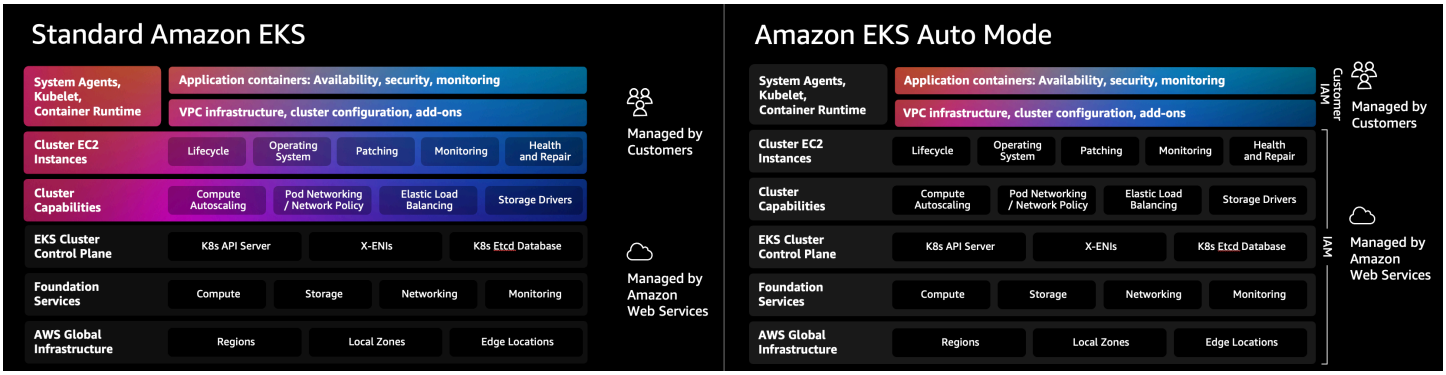
Amazon EKS 安全沉浸研討會

EKS Auto 模式 - 安全性

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

Amazon EKS Auto Mode 透過將 AWS 的安全管理延伸到控制平面之外來引入增強型安全功能，以包含工作者節點和核心叢集元件。此全方位的安全模型可協助組織維持強大的安全狀態，同時降低營運開銷。

共同責任模型 - EKS Auto 模式



EKS Auto 模式的主要安全性增強功能包括：

- 減少攻擊面的最低容器最佳化作業系統
- 透過 EC2 受管執行個體強制執行的安全最佳實務
- 使用強制性節點輪換的自動化安全修補程式管理
- 內建節點隔離和安全性界限
- 以最少的必要許可簡化 IAM 整合

EKS Auto Mode 預設會強制執行這些安全控制，協助組織滿足其安全和合規要求，同時簡化叢集操作。此方法符合defense-in-depth原則，在基礎設施、節點和工作負載層級提供多層安全控制。

> **Important**
>
> 雖然 EKS Auto Mode 提供增強型安全功能，但組織仍應在應用程式層實作適當的安全控制，並遵循在叢集上執行之工作負載的安全最佳實務。

安全架構

EKS Auto Mode 會跨 EKS 基礎設施的多層實作安全控制，從控制平面到個別節點。了解此架構對於有效管理和保護 EKS 叢集至關重要。

控制平面安全性

EKS Auto Mode 中的 EKS 控制平面維護與傳統 EKS 叢集相同的高安全標準，同時新增新的安全功能：

- 信封加密：所有 Kubernetes API 資料都會使用信封加密自動加密。
- KMS 整合：使用 AWS KMS 搭配 Kubernetes KMS 提供者 v2，搭配 AWS 擁有的金鑰或客戶受管金鑰 (CMK) 的選項。
- 增強型元件管理：自動擴展、ENI 管理和 EBS 控制器等關鍵元件會移至叢集之外，並由 AWS 管理。
- 改善安全控制和稽核功能：除了標準 EKS 叢集之外，EKS Auto Mode 所需的許可是透過叢集 IAM 角色而非個別節點角色進行完整管理。

IAM 整合與存取管理

EKS Auto Mode 透過 EKS 存取項目和 EKS Pod 身分，提供與 AWS Identity and Access Management (IAM) 的增強整合。

叢集存取管理

EKS Auto Mode 透過叢集存取管理 (CAM) API 引入了叢集存取管理的改進：

- 透過 標準化身分驗證模式 `EKS_API`
- 透過 API 型存取管理增強安全性
- 使用存取項目和存取政策簡化存取控制

您可以建立存取項目來管理叢集存取：

```
aws eks create-access-entry \
    --cluster-name ${EKS_CLUSTER_NAME} \
    --principal-arn arn:aws:iam::${ACCOUNT_ID}:role/${IAM_ROLE_NAME} \
    --type STANDARD
```

> **Important**
>
> 雖然仍然可以建立具有CONFIG_MAP_AND_API身分驗證模式的 EKS Auto Mode 叢集，但這不是標準方法，強烈建議對新叢集使用預設API身分驗證模式。相較於舊版 ConfigMap 型方法，API 型身分驗證可提供增強的安全性和簡化的存取管理。

EKS Pod 身分識別

EKS Auto Mode 隨附已部署的 Pod Identity Agent，可讓您以簡化的方式將 AWS IAM 許可授予 Pod：

- 在沒有 OIDC 提供者組態的情況下簡化 IAM 許可管理
- 與 IRSA 相比，降低了營運開銷
- 透過工作階段標記和 ABAC 支援增強安全性

```
aws eks create-pod-identity-association \
  --cluster-name ${EKS_CLUSTER_NAME} \
  --role-arn arn:aws:iam::${AWS_ACCOUNT_ID}:role/${IAM_ROLE_NAME} \
  --namespace ${NAMESPACE} \
  --service-account ${SERVICE_ACCOUNT_NAME}
```

> **Important**
>
> 與 IRSA 相比，Pod Identity 是 EKS Auto Mode 中 IAM 許可的建議方法，因為它提供增強的安全功能和簡化的管理。

節點 IAM 角色

EKS Auto Mode 使用新的 `AmazonEKSWorkerNodeMinimalPolicy` ，只提供 EKS Auto Mode 節點操作所需的許可。這些許可：

- 與傳統節點政策相比，提供一組減少的許可
- 遵守最低權限原則
- 由 AWS 自動管理和更新

這種最少的政策方法透過限制節點及其工作負載可用的許可，協助改善安全狀態。

節點安全性

EKS Auto Mode 在節點層級引入了幾項重大的安全改進：

EC2 受管執行個體安全性

EKS Auto Mode 節點使用具有增強型安全屬性的 Amazon EC2 受管執行個體：

- IAM 強制執行的限制，可防止可能影響 AWS 操作節點能力的操作
- 不可變的基礎設施模式，其中組態變更需要節點替換
- 在 21 天內進行強制性節點取代，以確保定期安全性更新
- 使用具有控制跳轉限制的 IMDSv2 限制存取執行個體中繼資料

作業系統安全

作業系統是 Bottlerocket 的自訂變體，針對 EKS Auto Mode 進行最佳化，其中包括：

- 唯讀根檔案系統
- 根據預設，SELinux 已啟用強制存取控制
- 使用唯一 SELinux MCS 標籤的自動 Pod 隔離
- 停用 SSH 存取和移除不必要的服務
- 透過節點輪換的自動化安全修補程式

節點元件安全性

節點元件設定了安全最佳實務：

- 使用安全預設值設定的 Kubelet
- 容器執行時間強化組態
- 自動化憑證管理和輪換

- 受限制node-to-control-plane通訊

網路安全

EKS Auto Mode 實作數個網路安全功能，以確保叢集內和外部資源的安全通訊：

VPC CNI 網路政策

EKS Auto Mode 利用 Amazon VPC CNI 外掛程式的原生 Kubernetes 網路政策支援：

- 與上游 Kubernetes 網路政策 API 整合
- 允許對 pod-to-pod精細控制
- 同時支援輸入和輸出規則

若要在 EKS Auto 模式中啟用網路政策支援，您需要使用`configMap`資訊清單設定 VPC CNI 附加元件。請見此處範例：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: amazon-vpc-cni
  namespace: kube-system
data:
  enable-network-policy: "true"
```

定義網路政策支援也需要在節點類別中設定，如下所示：

```
apiVersion: eks.amazonaws.com/v1
kind: NodeClass
metadata:
  name: example-node-class
spec:
  networkPolicy: DefaultAllow
  networkPolicyEventLogs: Enabled
```

啟用後，您可以建立網路政策來控制流量：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
```

```
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

增強型 ENI 管理

EKS Auto Mode 可改善彈性網路界面 (ENI) 管理的安全性：

- AWS 受管 ENI 連接和組態
- 控制流量與資料流量的分隔
- 自動化 IP 地址管理，減少節點上所需的權限

儲存體安全性

EKS Auto Mode 為暫時性和持久性儲存提供增強的安全功能：

暫時性儲存

- 寫入暫時性磁碟區的所有資料都會自動加密
- 使用業界標準的 AES-256 密碼編譯演算法
- 由服務無縫處理的加密和解密

EBS 磁碟區

- 根和資料 EBS 磁碟區一律加密
- 磁碟區設定為在執行個體終止時刪除
- 您可以選擇指定用於加密的自訂 KMS 金鑰

EFS 整合

- 支援使用 EFS 傳輸中的加密
- EFS 檔案系統的自動靜態加密
- 與 EFS 存取點整合以增強存取控制

> **Important**
>
> 搭配 EKS Auto Mode 使用 EFS 時，請確保在 EFS 檔案系統層級設定適當的加密設定，因為 EKS Auto Mode 不會直接管理 EFS 加密。

監控和記錄

EKS Auto Mode 提供增強的監控和記錄功能，可協助您維持叢集安全狀態和操作運作狀態的可見性。

控制平面記錄

EKS Auto Mode 會維護與標準 EKS 相同的控制平面記錄功能，但預設會啟用所有日誌以進行增強型監控。

- 日誌會傳送至 Amazon CloudWatch Logs
- 根據預設，EKS Auto Mode 會啟用所有控制平面日誌：API 伺服器、稽核、驗證器、控制器管理員和排程器
- EKS Auto Mode 可讓您詳細了解叢集操作和安全性事件

> **Important**
>
> 控制平面記錄會產生 CloudWatch 中日誌儲存的額外費用。請仔細考慮您的記錄策略，以平衡安全需求與成本管理。

節點層級記錄

EKS Auto Mode 可增強節點層級記錄：

- 系統日誌會自動收集，並可透過 CloudWatch Logs 存取
- 節點日誌即使在節點終止後仍會保留，以協助事件後分析
- 增強對節點層級安全性事件和操作問題的可見性

Amazon GuardDuty 整合

EKS Auto Mode 叢集與 Amazon GuardDuty 無縫整合，以增強威脅偵測。功能包括：

- 自動化掃描控制平面稽核日誌
- 可啟用工作負載監控的執行期監控
- 與現有 GuardDuty 調查結果和提醒機制整合

若要在 Amazon GuardDuty for Kubernetes 稽核日誌上啟用 EKS Auto Mode 保護，您可以執行下列命令：

```
aws guardduty update-detector \
    --detector-id 12abc34d567e8fa901bc2d34e56789f0 \
    --data-sources '{"Kubernetes":{"AuditLogs":{"Enable":true}}}'
```

執行期安全性的 Amazon GuardDuty 整合

Amazon GuardDuty 為 EKS Auto Mode 叢集提供基本的執行期安全監控，提供全面的威脅偵測和安全性監控功能。此整合特別重要，因為它有助於即時識別潛在的安全威脅和惡意活動。

EKS Auto Mode 的主要 GuardDuty 功能

- 執行期監控：
  - 持續監控執行時間行為
  - 偵測惡意或可疑活動
  - 識別潛在的容器逸出嘗試
  - 監控異常程序執行或網路連線
- Kubernetes 特定威脅偵測：
  - 識別可疑的 Pod 部署嘗試
  - 偵測遭入侵的容器
  - 監控特權容器啟動
  - 識別可疑的服務帳戶用量
- 全面的調查結果類型：
  - 政策：Kubernetes/* - 偵測違反安全最佳實務的行為
  - 影響：Kubernetes/* - 識別可能受影響的資源
  - 探索：Kubernetes/* - 偵測偵察活動
  - 執行：Kubernetes/* - 識別可疑的執行模式
  - 持續性：Kubernetes/* - 偵測潛在的持續性威脅

若要在 Amazon GuardDuty for Kubernetes 稽核日誌和執行期監控上啟用 EKS Auto Mode 保護，您可以執行下列命令：

```
aws guardduty update-detector \
    --detector-id 12abc34d567e8fa901bc2d34e56789f0 \
    --data-sources '{
        "Kubernetes": {
            "AuditLogs": {"Enable": true},
            "RuntimeMonitoring": {"Enable": true}
        }
    }'
```

> **Important**
>
> EKS Auto Mode 叢集會自動支援 GuardDuty 執行期監控，可在節點層級提供增強的安全性可見性，而無需額外的組態。

GuardDuty 調查結果整合

GuardDuty 調查結果可以與其他 AWS 服務整合，以進行自動化回應：

- EventBridge 規則：

```
{
  "source": ["aws.guardduty"],
  "detail-type": ["GuardDuty Finding"],
  "detail": {
    "type": ["Runtime:Container/*", "Runtime:Kubernetes/*"],
    "severity": [4, 5, 6, 7, 8]
  }
}
```

- Security Hub 整合：

```
# Enable Security Hub integration
aws securityhub enable-security-hub \
    --enable-default-standards \
    --tags '{"Environment":"Production"}' \
```

```
    --region us-west-2
```

使用 EKS Auto 模式的 GuardDuty 最佳實務

1. 啟用所有調查結果類型：
   - 同時啟用 Kubernetes 稽核日誌監控和執行期監控
   - 設定所有嚴重性層級的問題清單
2. 實作自動化回應：
   - 為高嚴重性問題清單建立 EventBridge 規則
   - 與 AWS Security Hub 整合，以進行集中式安全管理
   - 在適當的情況下設定自動修復動作
3. 定期檢閱和調校：
   - 定期檢閱 GuardDuty 調查結果
   - 根據您的環境調校偵測閾值
   - 根據新的問題清單類型更新回應程序
4. 跨帳戶管理：
   - 考慮使用 GuardDuty 管理員帳戶進行集中式管理
   - 啟用跨多個帳戶的調查結果彙總

> **Warning**
>
> 雖然 GuardDuty 提供全面的安全監控，但它應該是defense-in-depth策略的一部分，其中包括其他安全控制，例如網路政策、Pod 安全標準和適當的 RBAC 組態。

常見問答集 (FAQ)

問：EKS Auto Mode 在安全性方面與標準 EKS 有何不同？ 答：EKS Auto Mode 透過 EC2 受管執行個體、自動修補、強制節點輪換和內建安全控制提供增強的安全性。它透過讓 AWS 管理更多安全層面來減少營運開銷，同時保持強大的安全狀態。

問：我是否仍然可以使用現有的安全工具和政策搭配 EKS Auto Mode？ 答：是，EKS Auto Mode 與大多數現有的安全工具和政策相容。不過，由於 EKS Auto Mode 節點的受管性質，某些節點層級的安全工具可能需要調整。

問：如何在 EKS Auto 模式下部署安全代理程式和監控工具？ 答：在 EKS Auto 模式下，安全代理程式和監控工具應該部署為 Kubernetes 工作負載（通常是 DaemonSets，預設在每個節點上部署一個 Pod 執行個體），而不是直接安裝在節點作業系統上。此方法符合 EKS Auto Mode 的不可變基礎設施模型。範例：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: security-agent
  namespace: security
spec:
  selector:
    matchLabels:
      app: security-agent
  template:
    metadata:
      labels:
        app: security-agent
    spec:
      containers:
      - name: security-agent
        image: security-vendor/agent:latest
        securityContext:
          privileged: false
          # Use specific capabilities instead of privileged mode
          capabilities:
            add: ["NET_ADMIN", "SYS_ADMIN"]
```

問：第三方安全解決方案是否與 EKS Auto Mode 相容？ 答：許多熱門的第三方安全解決方案已更新為支援 EKS Auto Mode，但一律建議與您的安全廠商驗證特定版本和部署需求，因為 EKS Auto Mode 的支援可能需要更新版本或特定部署組態。

問：EKS Auto Mode 中的安全代理程式有哪些限制？ 答：金鑰限制包括：

- 無法直接存取修改節點的作業系統
- 節點輪換之間沒有持久性
- 必須與容器型部署相容
- 需要遵守節點不可變性
- 可能需要不同的權限組態
- 節點的任何持續變更都應透過 NodePools和 NodeClasses 資源完成。

> **Note**
>
> 雖然 EKS Auto Mode 可能需要調整您的安全工具部署策略，但這些變更通常會導致更可維護且安全的組態符合雲端原生最佳實務。EKS Auto Mode 預期會完全接管其管理的大部分功能。因此，如果您可以取得這些功能的任何手動變更，EKS Auto Mode 可能會覆寫或捨棄這些功能。

問：我可以搭配 EKS Auto Mode 使用自訂 AMIs 嗎？ 答：EKS Auto Mode 目前不支援自訂 AMIs。這是根據設計，因為 AWS 會在共同責任模型中管理節點的安全性、修補和維護。EKS Auto Mode 節點使用由 AWS 最佳化和維護的特殊 Bottlerocket 變體。

問：以 EKS Auto 模式自動輪換節點的頻率為何？ 答：EKS Auto Mode 中的節點的生命週期上限為 21 天。它們將在此限制之前自動取代，以確保定期安全性更新和修補程式應用程式。

問：SSH 是否可以進入 EKS Auto Mode 節點進行故障診斷？ 答：否，EKS Auto 模式不提供直接 SSH 存取。反之，您可以使用 NodeDiagnostic 自訂資源定義 (CRD) 來收集系統日誌和偵錯資訊。

問：根據預設，網路政策支援是否在 EKS Auto Mode 中啟用？ 答：目前，網路政策支援需要透過 VPC CNI 附加元件組態明確啟用。啟用後，您可以使用標準 Kubernetes 網路政策。

問：我應該將 IRSA 或 Pod Identity 與 EKS Auto Mode 搭配使用嗎？ 答：雖然兩者都支援，但 Pod Identity 是 EKS Auto Mode 的建議方法，因為它已包含 Pod Identity Security 代理程式附加元件，並提供增強的安全功能和簡化的管理。

問：我是否仍然可以在 EKS Auto 模式中使用 aws-auth ConfigMap？ 答：ConfigMap `aws-auth` 是已棄用的功能。建議使用以 API 為基礎的預設身分驗證方法，以增強安全性並簡化存取管理。

問：如何在 EKS Auto 模式下監控安全事件？ 答：EKS Auto Mode 與多個監控解決方案整合，包括 GuardDuty、CloudWatch 和 CloudTrail。GuardDuty 特別針對 EKS 工作負載提供增強的執行期安全監控。

問：如何從 EKS Auto Mode 節點收集日誌？ 答：使用 NodeDiagnostic CRD，它會自動將日誌上傳到 S3 儲存貯體。您也可以使用 CloudWatch Container Insights 和 AWS Distro for OpenTelemetry。

> **Note**
>
> 隨著新功能新增至 EKS Auto Mode，以及當我們收到客戶的常見問題時，此常見問答集區段會定期更新。

身分和存取權管理

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

Identity and Access Management (IAM) 是一種 AWS 服務，可執行兩項基本功能：身分驗證和授權。驗證涉及身分驗證，而授權會管理 AWS 資源可執行的動作。在 AWS 中，資源可以是另一個 AWS 服務，例如 EC2 或 AWS 委託人，例如 IAM 使用者或角色。管理允許資源執行之動作的規則會以 IAM 政策表示。

控制對 EKS 叢集的存取

Kubernetes 專案支援各種不同的策略來驗證 kube-apiserver 服務的請求，例如 Bearer Tokens、X.509 憑證、OIDC 等。EKS 目前支援 Webhook 字符身分驗證、服務帳戶字符，以及截至 2021 年 2 月 21 日的 OIDC 身分驗證。

Webhook 身分驗證策略會呼叫 Webhook，以驗證承載字符。在 EKS 上，當您執行`kubectl`命令時，這些承載字符是由 AWS CLI 或 aws-iam-authenticator 用戶端產生。當您執行命令時，權杖會傳遞至 kube-apiserver，再將其轉送至身分驗證 Webhook。如果請求格式正確，Webhook 會呼叫內嵌在字符內文中的預先簽章 URL。此 URL 會驗證請求的簽章，並將使用者的相關資訊傳回 kube-apiserver，例如使用者帳戶、Arn 和 UserId。

若要手動產生身分驗證字符，請在終端機視窗中輸入下列命令：

```
aws eks get-token --cluster-name <cluster_name> --region <region>
```

輸出應該像這樣：

```
{
    "kind": "ExecCredential",
    "apiVersion": "client.authentication.k8s.io/v1alpha1",
    "spec": {},
    "status": {
        "expirationTimestamp": "2024-12-20T17:38:48Z",
        "token": "k8s-aws-
v1.aHR0cHM6Ly9zdHMudXMtd2VzdC0yLmFtYXpvbmF3cy5jb20vP0FjdGlvbj1HZ...."
    }
}
```

您也可以以程式設計方式取得字符。以下是以 Go 撰寫的範例：

```
package main

import (
  "fmt"
  "log"
  "sigs.k8s.io/aws-iam-authenticator/pkg/token"
)

func main()  {
  g, _ := token.NewGenerator(false, false)
  tk, err := g.Get("<cluster_name>")
  if err != nil {
    log.Fatal(err)
  }
  fmt.Println(tk)
}
```

輸出應該像這樣：

```
{
  "kind": "ExecCredential",
  "apiVersion": "client.authentication.k8s.io/v1alpha1",
  "spec": {},
  "status": {
    "expirationTimestamp": "2020-02-19T16:08:27Z",
    "token": "k8s-aws-
v1.aHR0cHM6Ly9zdHMuYW1hem9uYXdzLmNvbS8_QWN0aW9uPUdldENhbGxlcklkZW50aXR5JlZlcnNpb249MjAxMS0wNi0x
  }
}
```

每個字符都以 開頭，k8s-aws-v1.後面接著 base64 編碼字串。字串解碼時，應類似類似如下：

```
https://sts.amazonaws.com/?Action=GetCallerIdentity&Version=2011-06-15&X-
Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=XXXXJPFRILKNSRC2W5QA
%2F20200219%2Fus-xxxx-1%2Fsts%2Faws4_request&X-Amz-Date=20200219T155427Z&X-
Amz-Expires=60&X-Amz-SignedHeaders=host%3Bx-k8s-aws-id&X-Amz-
Signature=XXXf8f3285e320ddb5e683a5c9a405301ad76546f24f28111fdad09cf648a393
```

權杖包含預先簽章的 URL，其中包含 Amazon 登入資料和簽章。如需其他詳細資訊，請參閱 https：// https://docs.aws.amazon.com/STS/latest/APIReference/API_GetCallerIdentity.html。

字符的存留時間 (TTL) 為 15 分鐘，之後需要產生新的字符。當您使用類似 的用戶端時，系統會自動處理此問題，kubectl不過，如果您使用 Kubernetes 儀表板，則需要產生新的字符，並在每次字符過期時重新驗證。

一旦 AWS IAM 服務已驗證使用者的身分，kube-apiserver 就會讀取kube-system命名空間中的 aws-auth ConfigMap，以決定要與使用者建立關聯的 RBAC 群組。ConfigMap aws-auth 用於在 IAM 主體之間建立靜態映射，即 IAM 使用者和角色，以及 Kubernetes RBAC 群組。您可以在 Kubernetes RoleBindings 或 ClusterRoleBindings 中參考 RBAC 群組。它們類似於 IAM 角色，它們定義一組動作 （動詞），可以針對 Kubernetes 資源 （物件） 集合執行。

CloudWatch 查詢可協助使用者識別將請求傳送至全域 STS 端點的用戶端

執行以下 CloudWatch 查詢以取得 sts 端點。如果 stsendpoint 等於 "sts.amazonaws.com"，則它是全域 STS 端點。如果 stsendpoint 等於 "sts.<region>.amazonaws.com"，則它是區域 STS 端點。

```
fields @timestamp, @message, @logStream, @log,stsendpoint
| filter @logStream like /authenticator/
| filter @message like /stsendpoint/
| sort @timestamp desc
| limit 10000
```

叢集存取管理員

Cluster Access Manager 現在是管理 AWS IAM 主體對 Amazon EKS 叢集的存取的偏好方式，是 AWS API 的功能，也是 EKS v1.23 和更新版本叢集 （全新或現有） 的選擇加入功能。它簡化了 AWS IAM 和 Kubernetes RBACs 之間的身分映射，無需在 AWS 和 Kubernetes APIs 之間切換或編輯 aws-auth ConfigMap 以進行存取管理、降低營運開銷，並協助解決設定錯誤。此工具也可讓叢集管理員撤銷或精簡自動授予用於建立叢集之 AWS IAM 主體的cluster-admin許可。

此 API 依賴兩個概念：

- 存取項目：直接連結至 AWS IAM 主體 （使用者或角色） 的叢集身分，允許對 Amazon EKS 叢集進行身分驗證。
- 存取政策： 是 Amazon EKS 特定政策，提供存取項目在 Amazon EKS 叢集中執行動作的授權。

在啟動時，Amazon EKS 僅支援預先定義和 AWS 受管政策。存取政策不是 IAM 實體，由 Amazon EKS 定義和管理。

Cluster Access Manager 允許將上游 RBAC 與支援允許和傳遞（但不拒絕）API 伺服器請求之 Kubernetes AuthZ 決策的存取政策結合。當上游 RBAC 和 Amazon EKS 授權方無法判斷請求評估的結果時，就會發生拒絕決策。

透過此功能，Amazon EKS 支援三種身分驗證模式：

1. CONFIG_MAP 以繼續只使用 aws-auth configMap。
2. API_AND_CONFIG_MAP 從 EKS 存取項目 APIs和 aws-auth configMap 中來源已驗證的 IAM 主體，優先考慮存取項目。將現有aws-auth許可遷移至存取項目的理想選擇。
3. API 完全依賴 EKS Access Entry APIs。這是新的建議方法。

若要開始使用，叢集管理員可以建立或更新 Amazon EKS 叢集、將偏好的身分驗證設定為 API_AND_CONFIG_MAP或 API方法，以及定義存取項目以授予所需的 AWS IAM 主體存取權。

```
$ aws eks create-cluster \
    --name <CLUSTER_NAME> \
    --role-arn <CLUSTER_ROLE_ARN> \
    --resources-vpc-config
 subnetIds=<value>,endpointPublicAccess=true,endpointPrivateAccess=true \
    --logging '{"clusterLogging":[{"types":
["api","audit","authenticator","controllerManager","scheduler"],"enabled":true}]}' \
    --access-config
 authenticationMode=API_AND_CONFIG_MAP,bootstrapClusterCreatorAdminPermissions=false
```

上述命令是建立 Amazon EKS 叢集的範例，該叢集已沒有叢集建立者的管理員許可。

您可以更新 Amazon EKS 叢集組態，以使用 update-cluster-config命令啟用 API authenticationMode，在現有叢集上使用 執行此操作CONFIG_MAP，您必須先更新至 API_AND_CONFIG_MAP，然後再更新至 API。這些操作無法還原，這表示無法從 切換API到 API_AND_CONFIG_MAP或 CONFIG_MAP，也無法從 切換API_AND_CONFIG_MAP到 CONFIG_MAP。

```
$ aws eks update-cluster-config \
    --name <CLUSTER_NAME> \
    --access-config authenticationMode=API
```

API 支援命令來新增和撤銷對叢集的存取，以及驗證指定叢集的現有存取政策和存取項目。系統會建立預設政策以符合 Kubernetes RBACs如下所示。

EKS 存取政策	Kubernetes RBAC
AmazonEKSClusterAdminPolicy	cluster-admin
AmazonEKSAdminPolicy	admin
AmazonEKSEditPolicy	編輯
AmazonEKSViewPolicy	檢視

```
$ aws eks list-access-policies
{
    "accessPolicies": [
        {
            "name": "AmazonEKSAdminPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSAdminPolicy"
        },
        {
            "name": "AmazonEKSClusterAdminPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy"
        },
        {
            "name": "AmazonEKSEditPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSEditPolicy"
        },
        {
            "name": "AmazonEKSViewPolicy",
            "arn": "arn:aws:eks::aws:cluster-access-policy/AmazonEKSViewPolicy"
        }
    ]
}

$ aws eks list-access-entries --cluster-name <CLUSTER_NAME>

{
    "accessEntries": []
}
```

在沒有叢集建立者管理員許可的情況下建立叢集時，沒有可用的存取項目，這是預設建立的唯一項目。

ConfigMap **aws-auth**（已棄用）

Kubernetes 與 AWS 身分驗證整合的其中一種方法是透過位於 kube-system 命名空間中的 aws-auth ConfigMap。它負責將 AWS IAM 身分（使用者、群組和角色）身分驗證映射至 Kubernetes 角色型存取控制 (RBAC) 授權。ConfigMap aws-auth 會在佈建階段自動在您的 Amazon EKS 叢集中建立。它最初是為了允許節點加入叢集而建立，但如上所述，您也可以使用此 ConfigMap 將 RBACs存取權新增至 IAM 主體。

若要檢查叢集的 aws-auth ConfigMap，您可以使用下列命令。

```
kubectl -n kube-system get configmap aws-auth -o yaml
```

這是 aws-auth ConfigMap 預設組態的範例。

```
apiVersion: v1
data:
  mapRoles: |
    - groups:
      - system:bootstrappers
      - system:nodes
      - system:node-proxier
      rolearn: arn:aws:iam::<AWS_ACCOUNT_ID>:role/kube-system-<SELF_GENERATED_UUID>
      username: system:node:{{SessionName}}
kind: ConfigMap
metadata:
  creationTimestamp: "2023-10-22T18:19:30Z"
  name: aws-auth
  namespace: kube-system
```

此 ConfigMap 的主要工作階段位於 mapRoles區塊data的 下，基本上由 3 個參數組成。

- 群組：要映射 IAM 角色的 Kubernetes 群組。這可以是預設群組，或在 clusterrolebinding或中指定的自訂群組rolebinding。在上述範例中，我們只宣告了系統群組。
- rolearn：使用下列格式 ，將 AWS IAM 角色的 ARN 映射至 Kubernetes 群組/群組新增arn:<PARTITION>:iam::<AWS_ACCOUNT_ID>:role/role-name。
- username：Kubernetes 中對應至 AWS IAM 角色的使用者名稱。這可以是任何自訂名稱。

您也可以映射 AWS IAM 使用者的許可，在 aws-authConfigMap data的 mapUsers下定義 的新組態區塊，取代 userarn 的 rolearn 參數，但作為最佳實務，一律建議使用者mapRoles使用。

若要管理許可，您可以編輯 aws-auth ConfigMap 新增或移除對 Amazon EKS 叢集的存取。雖然可以手動編輯 aws-auth ConfigMap，但建議使用 等工具eksctl，因為這是非常敏感的組態，不正確的組態可能會將您鎖定在 Amazon EKS 叢集之外。如需詳細資訊，請參閱以下子節 使用工具對 aws-auth ConfigMap 進行變更。

ConfigMap 型存取管理的優勢

1. 降低組態錯誤的風險：直接 API 型管理可消除與手動 ConfigMap 編輯相關的常見錯誤。這有助於防止意外刪除或語法錯誤，這些錯誤可能會鎖定使用者離開叢集。
2. 增強型最低權限原則：從叢集建立者身分移除叢集管理員許可的需求，並允許更精細且適當的許可指派。您可以選擇為打破玻璃的使用案例新增此許可。
3. 增強型安全模型：在套用存取項目之前，提供存取項目的內建驗證。此外， 提供更緊密的 AWS IAM 整合以進行身分驗證。
4. 簡化操作：提供更直覺的方式，透過 AWS 原生工具管理許可。

叢集存取建議

結合 IAM Identity Center 與 CAM API

- 簡化管理：透過使用叢集存取管理 API 搭配 IAM Identity Center，管理員可以與其他 AWS 服務一起管理 EKS 叢集存取，減少在不同介面之間切換或手動編輯 ConfigMaps 的需求。
- 使用存取項目，從叢集外部管理 IAM 主體的 Kubernetes 許可。您可以使用 EKS API、AWS 命令列界面、AWS SDKs來新增和管理對叢集的存取。 AWS CloudFormation 這意味著您可以使用建立叢集時使用的工具來管理使用者。
- 利用自動化，如本範例所示，以 AWS IAM Identity Center 做為具有 CAM API 做為進入點的 IdP 部署叢集。
- 精細 Kubernetes 許可可以透過存取項目和存取政策，與與 SSO 身分相關聯的 IAM 主體映射 Kubernetes 使用者或群組套用。
- 若要開始使用，請依照變更身分驗證模式來使用存取項目，然後遷移現有的 aws-auth ConfigMap 項目來存取項目。

將 EKS 叢集端點設為私有

根據預設，當您佈建 EKS 叢集時，API 叢集端點會設為公有，也就是可以從網際網路存取。儘管可從網際網路存取，但端點仍被視為安全，因為它需要 IAM 驗證所有 API 請求，然後由 Kubernetes RBAC

授權。也就是說，如果您的公司安全政策要求您限制從網際網路存取 API，或防止您將流量路由到叢集 VPC 外部，您可以：

- 將 EKS 叢集端點設定為私有。如需本主題的詳細資訊，請參閱修改叢集端點存取。
- 將叢集端點保持為公有，並指定哪些 CIDR 區塊可與叢集端點通訊。這些區塊實際上是允許存取叢集端點的一組列入允許清單的公有 IP 地址。
- 使用一組列入允許清單的 CIDR 區塊設定公開存取，並將私有端點存取設定為已啟用。這將允許從特定範圍的公有 IPs 公開存取，同時透過在佈建控制平面時佈建至叢集 VPC 的跨帳戶 ENIs，強制 kubelet（工作者）和 Kubernetes API 之間的所有網路流量。

請勿使用服務帳戶字符進行身分驗證

服務帳戶字符是長期的靜態登入資料。如果遭到入侵、遺失或遭竊，攻擊者可能可以執行與該字符相關聯的所有動作，直到刪除服務帳戶為止。有時，您可能需要為必須從叢集外部取用 Kubernetes API 的應用程式授予例外狀況，例如 CI/CD 管道應用程式。如果此類應用程式在 AWS 基礎設施上執行，例如 EC2 執行個體，請考慮使用執行個體描述檔並將其映射至 Kubernetes RBAC 角色。

使用對 AWS 資源的最低權限存取

IAM 使用者不需要為 AWS 資源指派權限，即可存取 Kubernetes API。如果您需要授予 IAM 使用者對 EKS 叢集的存取權，請在 ConfigMap `aws-auth` 中為該使用者建立對應至特定 Kubernetes RBAC 群組的項目。

從叢集建立者主體移除叢集管理員許可

根據預設，Amazon EKS 叢集會使用繫結至叢集建立者主體的永久`cluster-admin`許可來建立。使用 Cluster Access Manager API 時，您可以建立叢集`false`，而不需要此許可，即可在使用 `API_AND_CONFIG_MAP`或 API 身分驗證模式時將 `--access-config bootstrapClusterCreatorAdminPermissions` 設定為 。撤銷此存取權視為最佳實務，以避免對叢集組態進行任何不必要的變更。撤銷此存取權的程序遵循相同的程序來撤銷對叢集的任何其他存取權。

API 可讓您靈活地僅取消 IAM 主體與存取政策的關聯，在此情況下為`AmazonEKSClusterAdminPolicy`。

```
$ aws eks list-associated-access-policies \
    --cluster-name <CLUSTER_NAME> \
    --principal-arn <IAM_PRINCIPAL_ARN>
```

```
$ aws eks disassociate-access-policy --cluster-name <CLUSTER_NAME> \
    --principal-arn <IAM_PRINCIPAL_ARN. \
    --policy-arn arn:aws:eks::aws:cluster-access-policy/AmazonEKSClusterAdminPolicy
```

或完全移除與cluster-admin許可相關聯的存取項目。

```
$ aws eks list-access-entries --cluster-name <CLUSTER_NAME>

{
    "accessEntries": []
}

$ aws eks delete-access-entry --cluster-name <CLUSTER_NAME> \
  --principal-arn <IAM_PRINCIPAL_ARN>
```

在無法存取叢集的事件、緊急或中斷玻璃案例期間，如有需要，可以再次授予此存取權。

如果叢集仍使用CONFIG_MAP身分驗證方法設定，則所有額外的使用者都應該透過 aws-auth ConfigMap 授予叢集的存取權，並且在設定 aws-auth ConfigMap 之後，指派給建立叢集之實體的角色可以刪除，並且只有在發生事件、緊急或中斷玻璃情況，或 aws-authConfigMap 損毀且叢集無法存取時，才能重新建立。這在生產叢集中特別有用。

當多個使用者需要對叢集的相同存取權時，請使用 IAM 角色

不為每個個別 IAM 使用者建立項目，而是允許這些使用者擔任 IAM 角色，並將該角色映射至 Kubernetes RBAC 群組。這將更容易維護，尤其是隨著需要存取的使用者數量增加。

> **Important**
>
> 使用 ConfigMap 映射的 IAM aws-auth 實體存取 EKS 叢集時，所描述的使用者名稱會記錄在 Kubernetes 稽核日誌的使用者欄位中。如果您使用 IAM 角色，則不會記錄擔任該角色的實際使用者，也無法稽核。

如果仍然使用 aws-auth configMap 做為身分驗證方法，則在將 K8s RBAC 許可指派給 IAM 角色時，您應該在使用者名稱中包含 \{{SessionName}}。如此一來，稽核日誌將記錄工作階段名稱，以便您可以追蹤實際使用者擔任此角色的人員以及 CloudTrail 日誌。

```
- rolearn: arn:aws:iam::XXXXXXXXXXXX:role/testRole
```

```
  username: testRole:{{SessionName}}
  groups:
    - system:masters
```

> 在 Kubernetes 1.20 及更高版本中，由於 `user.extra.sessionName.0` 已新增至 Kubernetes 稽核日誌，因此不再需要此變更。

建立 RoleBindings 和 ClusterRoleBindings 時採用最低權限存取

如同先前授予對 AWS 資源的存取權，RoleBindings 和 ClusterRoleBindings 應僅包含執行特定函數所需的一組許可。除非絕對必要，否則請避免在角色和 ClusterRoles `["*"]`中使用 。如果您不確定要指派哪些許可，請考慮使用 audit2rbac 之類的工具，根據 Kubernetes 稽核日誌中觀察到的 API 呼叫自動產生角色和繫結。

使用自動化程序建立叢集

如先前步驟所示，建立 Amazon EKS 叢集時，如果未使用 `API_AND_CONFIG_MAP`或 API 身分驗證模式，且未選擇退出將`cluster-admin`許可委派給叢集建立者，則會自動在叢集的 RBAC 組態中授予`system:masters`許可給 IAM 實體使用者或角色，例如建立叢集的聯合身分使用者。即使是移除此許可的最佳實務，如此處所述，如果使用身分`CONFIG_MAP`驗證方法依賴 `aws-auth` ConfigMap，則無法撤銷此存取權。因此，最好使用連接到專用 IAM 角色的基礎設施自動化管道建立叢集，而沒有其他使用者或實體擔任的許可，並定期稽核此角色的許可、政策，以及有權觸發管道的人員。此外，此角色不應用於在叢集上執行例行動作，並僅用於透過 SCM 程式碼變更等管道觸發的叢集層級動作。

使用專用 IAM 角色建立叢集

當您建立 Amazon EKS 叢集時，IAM 實體使用者或角色，例如建立叢集的聯合身分使用者，會在叢集的 RBAC 組態中自動授予`system:masters`許可。此存取權無法移除，也不會透過 ConfigMap `aws-auth` 進行管理。因此，最好使用專用 IAM 角色建立叢集，並定期稽核誰可以擔任此角色。此角色不應用於在叢集上執行例行動作，而是應為此目的透過 ConfigMap `aws-auth` 授予其他使用者對叢集的存取權。設定 `aws-auth` ConfigMap 後，角色應受到保護，且僅在暫時提升權限模式/中斷玻璃中使用，適用於無法以其他方式存取叢集的情況。這在未設定直接使用者存取權的叢集中特別有用。

定期稽核對叢集的存取

需要存取的人員可能會隨著時間而變更。計劃定期稽核 `aws-auth` ConfigMap，以查看誰已獲授予存取權，以及他們獲指派的權利。您也可以使用開放原始碼工具，例如 kubectl-who-can 或 rbac-lookup 來檢查繫結至特定服務帳戶、使用者或群組的角色。進入稽核章節時，我們將進一步探索本主題。您可以在 NCC 群組的這篇文章中找到其他想法。

如果依賴 **aws-auth** configMap，請使用工具進行變更

格式不正確的 aws-auth ConfigMap 可能會導致您無法存取叢集。如果您需要變更 ConfigMap，請使用工具。

eksctl `eksctl`CLI 包含一個命令，用於將身分映射新增至 aws-auth ConfigMap。

檢視 CLI 說明：

```
$ eksctl create iamidentitymapping --help
...
```

檢查映射至 Amazon EKS 叢集的身分。

```
$ eksctl get iamidentitymapping --cluster $CLUSTER_NAME --region $AWS_REGION
ARN                                                                    USERNAME
               GROUPS                                                   ACCOUNT
arn:aws:iam::788355785855:role/kube-system-<SELF_GENERATED_UUID>      system:node:
{{SessionName}}     system:bootstrappers,system:nodes,system:node-proxier
```

將 IAM 角色設為叢集管理員：

```
$ eksctl create iamidentitymapping --cluster  <CLUSTER_NAME> --region=<region> --arn
 arn:aws:iam::123456:role/testing --group system:masters --username admin
...
```

如需詳細資訊，請參閱 [eksctl 文件]

keikoproj 的 [aws-auth]

`aws-auth` 作者 keikoproj 包含 cli 和 go 程式庫。

下載並檢視說明 CLI 說明：

```
$ go get github.com/keikoproj/aws-auth
...
$ aws-auth help
...
```

或者，`aws-auth`使用適用於 kubectl 的 [krew 外掛程式管理員]安裝 。

```
$ kubectl krew install aws-auth
```

```
...
$ kubectl aws-auth
...
```

如需詳細資訊，請參閱 GitHub 上的 aws-auth 文件，包括 go 程式庫。

AWS IAM Authenticator CLI

`aws-iam-authenticator` 專案包含用於更新 ConfigMap 的 CLI。

在 GitHub 下載版本。

將叢集許可新增至 IAM 角色：

```
$ ./aws-iam-authenticator add role --rolearn arn:aws:iam::185309785115:role/lil-dev-
role-cluster --username lil-dev-user --groups system:masters --kubeconfig ~/.kube/
config
...
```

驗證和存取管理的替代方法

雖然 IAM 是對需要存取 EKS 叢集的使用者進行身分驗證的首選方式，但可以使用身分驗證代理和 Kubernetes 模擬來使用 OIDC 身分提供者，例如 GitHub。兩個這類解決方案的文章已發佈在 AWS Open Source 部落格上：

- 透過 Teleport 使用 GitHub 登入資料向 EKS 驗證
- 使用 kube-oidc-proxy 跨多個 EKS 叢集進行一致的 OIDC 身分驗證

> **Important**
>
> EKS 原生支援不使用代理的 OIDC 身分驗證。如需詳細資訊，請閱讀啟動部落格簡介 Amazon EKS 的 OIDC 身分提供者身分驗證。如需示範如何使用 Dex 設定 EKS 的範例，Dex 是具有各種不同截斷方法連接器的熱門開放原始碼 OIDC 供應商，請參閱使用 Dex & dex-k8s-authenticator 向 Amazon EKS 驗證。如部落格所述，由 OIDC 供應商驗證的使用者名稱/使用者群組會顯示在 Kubernetes 稽核日誌中。

您也可以使用 AWS SSO 與外部身分提供者聯合 AWS，例如 Azure AD。如果您決定使用此功能，AWS CLI 2.0 版包含建立具名設定檔的選項，可讓您輕鬆地將 SSO 工作階段與目前的 CLI 工作階

段建立關聯，並擔任 IAM 角色。知道您在執行 之前必須擔任角色，kubectl因為 IAM 角色用於判斷使用者的 Kubernetes RBAC 群組。

EKS Pod 的身分和登入資料

在 Kubernetes 叢集內執行的某些應用程式需要許可才能呼叫 Kubernetes API 才能正常運作。例如，AWS Load Balancer 控制器需要能夠列出服務的端點。控制器也需要能夠叫用 AWS APIs 來佈建和設定 ALB。在本節中，我們將探索將權利和權限指派給 Pod 的最佳實務。

Kubernetes 服務帳戶

服務帳戶是一種特殊類型的物件，可讓您將 Kubernetes RBAC 角色指派給 Pod。系統會自動為叢集內的每個命名空間建立預設服務帳戶。當您在不參考特定服務帳戶的情況下將 Pod 部署到命名空間時，該命名空間的預設服務帳戶會自動指派給 Pod，而秘密，即該服務帳戶的 服務帳戶 (JWT) 字符，會掛載為 中的磁碟區/var/run/secrets/kubernetes.io/serviceaccount。在該目錄中解碼服務帳戶字符會顯示下列中繼資料：

```
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "default",
  "kubernetes.io/serviceaccount/secret.name": "default-token-5pv4z",
  "kubernetes.io/serviceaccount/service-account.name": "default",
  "kubernetes.io/serviceaccount/service-account.uid":
 "3b36ddb5-438c-11ea-9438-063a49b60fba",
  "sub": "system:serviceaccount:default:default"
}
```

預設服務帳戶具有 Kubernetes API 的下列許可。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2020-01-30T18:13:25Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:discovery
  resourceVersion: "43"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterroles/system%3Adiscovery
  uid: 350d2ab8-438c-11ea-9438-063a49b60fba
```

```
rules:
- nonResourceURLs:
  - /api
  - /api/*
  - /apis
  - /apis/*
  - /healthz
  - /openapi
  - /openapi/*
  - /version
  - /version/
  verbs:
  - get
```

此角色授權未驗證和已驗證的使用者讀取 API 資訊，並視為可公開存取。

當 Pod 內執行的應用程式呼叫 Kubernetes APIs 時，Pod 需要指派一個服務帳戶，明確授予它呼叫這些 APIs許可。與使用者存取的指導方針類似，繫結至服務帳戶的 角色或 ClusterRole 應該僅限於應用程式運作所需的 API 資源和方法，而不是其他方法。若要使用非預設服務帳戶，只需將 Pod `spec.serviceAccountName` 的欄位設定為您要使用的服務帳戶名稱。如需建立服務帳戶的詳細資訊，請參閱 https：//https://kubernetes.io/docs/reference/access-authn-authz/rbac/#service-account-permissions。

> **Note**
>
> 在 Kubernetes 1.24 之前，Kubernetes 會自動為每個服務帳戶建立秘密。此秘密掛載到位於 /var/run/secrets/kubernetes.io/serviceaccount 的 Pod，並由 Pod 用來向 Kubernetes API 伺服器進行身分驗證。在 Kubernetes 1.24 中，服務帳戶權杖會在 Pod 執行時動態產生，且預設僅在一小時內有效。將不會建立服務帳戶的秘密。如果您的應用程式在叢集之外執行，需要向 Kubernetes API 進行身分驗證，例如 Jenkins，您將需要建立 類型的秘密`kubernetes.io/service-account-token`，以及參考服務帳戶的註釋，例如 `metadata.annotations.kubernetes.io/service-account.name: <SERVICE_ACCOUNT_NAME>`。以這種方式建立的秘密不會過期。

服務帳戶的 IAM 角色 (IRSA)

IRSA 是一種功能，可讓您將 IAM 角色指派給 Kubernetes 服務帳戶。它的運作方式是利用稱為服務帳戶字符磁碟區投影的 Kubernetes 功能。使用參考 IAM 角色的服務帳戶設定 Pod 時，Kubernetes API 伺服器會在啟動時呼叫叢集的公有 OIDC 探索端點。端點以密碼編譯方式簽署 Kubernetes 發行

的 OIDC 權杖，並將產生的權杖掛載為磁碟區。此簽章權杖可讓 Pod 呼叫與 IAM 角色相關聯的 AWS APIs。叫用 AWS API 時，AWS SDKs會呼叫 `sts:AssumeRoleWithWebIdentity`。驗證權杖的簽章後，IAM 會交換 Kubernetes 發行的權杖以取得臨時 AWS 角色登入資料。

使用 IRSA 時，請務必重複使用 AWS 開發套件工作階段，以避免對 AWS STS 進行不必要的呼叫。

解碼 IRSA 的 (JWT) 字符會產生類似以下範例的輸出：

```
{
  "aud": [
    "sts.amazonaws.com"
  ],
  "exp": 1582306514,
  "iat": 1582220114,
  "iss": "https://oidc.eks.us-west-2.amazonaws.com/id/
D43CF17C27A865933144EA99A26FB128",
  "kubernetes.io": {
    "namespace": "default",
    "pod": {
      "name": "alpine-57b5664646-rf966",
      "uid": "5a20f883-5407-11ea-a85c-0e62b7a4a436"
    },
    "serviceaccount": {
      "name": "s3-read-only",
      "uid": "a720ba5c-5406-11ea-9438-063a49b60fba"
    }
  },
  "nbf": 1582220114,
  "sub": "system:serviceaccount:default:s3-read-only"
}
```

此特定字符會透過擔任 IAM 角色，將 Pod 檢視限定權限授予 S3。當應用程式嘗試從 S3 讀取時，字符會交換一組類似以下的臨時 IAM 登入資料：

```
{
    "AssumedRoleUser": {
        "AssumedRoleId": "AROA36C6WWEJULFUYMPB6:abc",
        "Arn": "arn:aws:sts::123456789012:assumed-role/eksctl-winterfell-addon-
iamserviceaccount-de-Role1-1D61LT75JH3MB/abc"
    },
    "Audience": "sts.amazonaws.com",
    "Provider": "arn:aws:iam::123456789012:oidc-provider/oidc.eks.us-
west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128",
```

```
    "SubjectFromWebIdentityToken": "system:serviceaccount:default:s3-read-only",
    "Credentials": {
        "SecretAccessKey": "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
        "SessionToken": "FwoGZXIvYXdzEGMaDMLxAZkuLpmSwYXShiL9A1S0X87VBC1mHCrRe/
pB2oesl1eXxUYnPJyC9ayOoXMvqXQsomq0xs6OqZ3vaa5Iw1HIyA4Cv1suLaOCoU3hNvOIJ6C94H1vU0siQYk7DIq9Av5RZ
Zv2A6zp5xGz9cWj2f0aD9v66vX4bexOs5t/YYhwuwAvkkJPSIGvxja0xRThnceHyFHKtj0Hbi/
PWAtlI8YJcDX69cM30JAHDdQHltm/4scFptW1hlvMaPWReCAaCrsHrATyka7ttw5YlUyvZ8EPogj6fwHlxmrXM9h1Bqdiko
LgkWsCTGlYcY8z3zkigJMbYn07ewTL5Ss7LazTJJa758I7PZan/
v3xQHd5DEc5WBneiV3iOznDFgup0VAMkIviVjVCkszaPSVEdK2NU7jtrh6Jfm7bU/3P6ZGCkyDLIa8MBn9KPXeJd/
yjTk5IifIwO/mDpGNUribg6TPxhzZ8b/XdZO1kS1gVgqjXyVCM+BRBh6C4H21w/eMzjCtDIpoxt5rGKL6Nu/
IFMipoC4fgx6LIIHwtGYMG7SWQi7OsMAkiwZRg0n68/RqWgLzBt/4pfjSRYuk=",
        "Expiration": "2020-02-20T18:49:50Z",
        "AccessKeyId": "ASIAIOSFODNN7EXAMPLE"
    }
}
```

作為 EKS 控制平面一部分執行的變動 Webhook 會將 AWS 角色 ARN 和 Web 身分字符檔案的路徑注入 Pod 做為環境變數。這些值也可以手動提供。

```
AWS_ROLE_ARN=arn:aws:iam::AWS_ACCOUNT_ID:role/IAM_ROLE_NAME
AWS_WEB_IDENTITY_TOKEN_FILE=/var/run/secrets/eks.amazonaws.com/serviceaccount/token
```

kubelet 會在超過其總 TTL 的 80% 時或在 24 小時後自動輪換投影字符。AWS SDKs負責在權杖輪換時重新載入權杖。如需 IRSA 的詳細資訊，請參閱 https：//https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts-technical-overview.html。

EKS Pod 身分識別

EKS Pod 身分是在 re：Invent 2023 上啟動的一項功能，可讓您將 IAM 角色指派給 kubernetes 服務帳戶，而無需為 AWS 帳戶中的每個叢集設定 Open Id Connect (OIDC) 身分提供者 (IDP)。若要使用 EKS Pod Identity，您必須在每個合格的工作者節點上部署以 DaemonSet Pod 身分執行的代理程式。此代理程式以 EKS 附加元件的形式提供給您，是使用 EKS Pod 身分功能的先決條件。您的應用程式必須使用支援的 AWS 開發套件版本，才能使用此功能。

為 Pod 設定 EKS Pod 身分時，EKS 會在 掛載和重新整理 Pod 身分字符`/var/run/secrets/pods.eks.amazonaws.com/serviceaccount/eks-pod-identity-token`。AWS 開發套件將使用此字符與 EKS Pod 身分代理程式通訊，該代理程式使用 Pod 身分字符和代理程式的 IAM 角色，透過呼叫 AssumeRoleForPodIdentity API 來為您的 Pod 建立臨時憑證。交付至 Pod 的 Pod 身分字符是從 EKS 叢集發出的 JWT，並以密碼編譯方式簽署，具有適當的 JWT 宣告，可與 EKS Pod 身分搭配使用。

若要進一步了解 EKS Pod 身分，請參閱此部落格。

您不需要對應用程式程式碼進行任何修改，即可使用 EKS Pod 身分。支援的 AWS 開發套件版本會使用登入資料提供者鏈結，自動探索 EKS Pod 身分提供的登入資料。如同 IRSA，EKS Pod 身分會在您的 Pod 中設定變數，引導他們如何尋找 AWS 登入資料。

針對 EKS Pod 身分使用 IAM 角色

- EKS Pod 身分只能直接擔任與 EKS 叢集屬於相同 AWS 帳戶的 IAM 角色。若要存取另一個 AWS 帳戶中的 IAM 角色，您必須在 SDK 組態或應用程式的程式碼中設定設定檔，以擔任該角色。 https://docs.aws.amazon.com/IAM/latest/UserGuide/sts_example_sts_AssumeRole_section.html
- 為服務帳戶設定 EKS Pod 身分時，設定 Pod 身分關聯的人員或程序必須具有該角色`iam:PassRole`的權限。
- 每個服務帳戶只能有一個透過 EKS Pod 身分與其相關聯的 IAM 角色，但您可以將相同的 IAM 角色與多個服務帳戶建立關聯。
- 與 EKS Pod 身分搭配使用的 IAM 角色必須允許`pods.eks.amazonaws.com`服務主體擔任這些角色，並設定工作階段標籤。以下是允許 EKS Pod 身分使用 IAM 角色的範例角色信任政策：

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "pods.eks.amazonaws.com"
      },
      "Action": [
        "sts:AssumeRole",
        "sts:TagSession"
      ],
      "Condition": {
        "StringEquals": {
          "aws:SourceOrgId": "${aws:ResourceOrgId}"
        }
      }
    }
  ]
}
```

AWS 建議使用 等條件金鑰aws:SourceOrgId，協助防範跨服務混淆代理人問題。在上述範例角色信任政策中， ResourceOrgId是等於 AWS 帳戶所屬 AWS 組織的 AWS Organizations Organization ID 的變數。EKS 將傳入的值aws:SourceOrgId等於擔任具有 EKS Pod 身分的角色時的值。

ABAC 和 EKS Pod 身分

當 EKS Pod 身分擔任 IAM 角色時，它會設定下列工作階段標籤：

EKS Pod 身分工作階段標籤	Value
kubernetes-namespace	與 EKS Pod 身分相關聯的 Pod 執行的命名空間。
kubernetes-service-account	與 EKS Pod 身分相關聯的 kubernetes 服務帳戶名稱
eks-cluster-arn	EKS 叢集的 ARN，例如 。 arn:${Partition}:eks:${Region}:${Account}:cluster/${ClusterName} 叢集 ARN 是唯一的，但如果在相同 AWS 帳戶中使用相同名稱刪除並重新建立叢集，則會有相同的 ARN。
eks-cluster-name	EKS 叢集的名稱。請注意，EKS 叢集名稱在您的 AWS 帳戶中可以相同，在其他 AWS 帳戶中也可以是 EKS 叢集。
kubernetes-pod-name	EKS 中 Pod 的名稱。
kubernetes-pod-uid	EKS 中 Pod 的 UID。

這些工作階段標籤可讓您使用屬性型存取控制 (ABAC)，僅將對 AWS 資源的存取權授予特定 kubernetes 服務帳戶。這樣做時，請務必了解 kubernetes 服務帳戶在命名空間中是唯一的，而 kubernetes 命名空間在 EKS 叢集中是唯一的。您可以使用 aws:PrincipalTag/<tag-key>全域條件金鑰，在 AWS 政策中存取這些工作階段標籤，例如 aws:PrincipalTag/eks-cluster-arn

例如，如果您只想授予特定服務帳戶的存取權，以使用 IAM 或資源政策存取您帳戶中的 AWS 資源，則需要檢查 eks-cluster-arn 和 kubernetes-namespace 標籤以及 ，kubernetes-

service-account以確保只有來自預期叢集的服務帳戶可以存取該資源，因為其他叢集可以具有相同的 kubernetes-service-accounts和 kubernetes-namespaces。

此範例 S3 儲存貯體政策只會授予其連接之 S3 儲存貯體中物件的存取權，前提是 kubernetes-service-account、 kubernetes-namespaceeks-cluster-arn全都符合其預期值，其中 EKS 叢集託管於 AWS 帳戶 111122223333。

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:root"
            },
            "Action": "s3:*",
            "Resource":            [
                "arn:aws:s3:::ExampleBucket/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalTag/kubernetes-service-account": "s3objectservice",
                    "aws:PrincipalTag/eks-cluster-arn": "arn:aws:eks:us-
west-2:111122223333:cluster/ProductionCluster",
                    "aws:PrincipalTag/kubernetes-namespace": "s3datanamespace"
                }
            }
        }
    ]
}
```

相較於 IRSA 的 EKS Pod 身分

EKS Pod 身分和 IRSA 都是將臨時 AWS 登入資料交付至 EKS Pod 的偏好方式。除非您有 IRSA 的特定使用案例，否則我們建議您在使用 EKS 時使用 EKS Pod 身分。此資料表有助於比較這兩個功能。

#	EKS Pod 身分識別	IRSA
需要在您的 AWS 帳戶中建立 OIDC IDP 的許可？	否	是

#	EKS Pod 身分識別	IRSA
每個叢集需要唯一的 IDP 設定	否	是
設定與 ABAC 搭配使用的相關工作階段標籤	是	否
需要 iam：PassRole 檢查嗎？	是	否
從您的 AWS 帳戶使用 AWS STS Quota？	否	是
可以存取其他 AWS 帳戶	間接使用角色鏈結	直接使用 sts：AssumeRoleWithWebIdentity
與 AWS SDKs相容	是	是
節點上需要 Pod Identity Agent Daemonset？	是	否

EKS Pod 建議的身分和登入資料

更新 aws-node 協助程式集以使用 IRSA

目前，aws-node 協助程式集已設定為使用指派給 EC2 執行個體的角色，將 IPs 指派給 Pod。此角色包含數個 AWS 受管政策，例如 AmazonEKS_CNI_Policy 和 EC2ContainerRegistryReadOnly，可有效允許節點上執行的所有 Pod 連接/分離 ENIs、指派/取消指派 IP 地址，或從 ECR 提取映像。由於這會對您的叢集造成風險，建議您更新 aws-node 協助程式集以使用 IRSA。您可以在本指南的 儲存庫中找到執行此操作的指令碼。

aws-node 協助程式集支援 1.15.5 版和更新版本的 EKS Pod 身分。

限制存取指派給工作者節點的執行個體描述檔

當您使用 IRSA 或 EKS Pod 身分時，它會先更新 Pod 的登入資料鏈以使用 IRSA 或 EKS Pod 身分，不過，Pod 仍然可以繼承指派給工作者節點的執行個體描述檔權利。對於不需要這些許可的 Pod，您可以封鎖對執行個體中繼資料的存取，以協助確保您的應用程式只有其所需的許可，而不是其節點。

> **Warning**
>
> 封鎖對執行個體中繼資料的存取，可防止不使用 IRSA 或 EKS Pod 身分的 Pod 繼承指派給工作者節點的角色。

您可以封鎖對執行個體中繼資料的存取，方法是要求執行個體僅使用 IMDSv2，並將跳轉計數更新為 1，如以下範例所示。您也可以在節點群組的啟動範本中包含這些設定。請勿停用執行個體中繼資料，因為這會使節點終止處理常式等元件和其他依賴執行個體中繼資料的物件無法正常運作。

```
$ aws ec2 modify-instance-metadata-options --instance-id <value> --http-tokens required
 --http-put-response-hop-limit 1
...
```

如果您使用 Terraform 建立用於受管節點群組的啟動範本，請新增中繼資料區塊來設定跳轉計數，如此程式碼片段所示：

```
tf hl_lines="7" resource "aws_launch_template" "foo" { name = "foo" …
 metadata_options { http_endpoint = "enabled" http_tokens = "required"
http_put_response_hop_limit = 1 instance_metadata_tags = "enabled" } …
```

您也可以在節點上操作 iptable，以封鎖 Pod 對 EC2 中繼資料的存取。如需此方法的詳細資訊，請參閱限制對執行個體中繼資料服務的存取。

如果您有使用舊版 AWS 開發套件的應用程式不支援 IRSA 或 EKS Pod 身分，您應該更新 SDK 版本。

將 IRSA 角色的 IAM 角色信任政策範圍限定為服務帳戶名稱、命名空間和叢集

信任政策的範圍可以是命名空間或命名空間內的特定服務帳戶。使用 IRSA 時，最好包含服務帳戶名稱，讓角色信任政策盡可能明確。這將有效地防止相同命名空間中的其他 Pod 擔任該角色。當您使用 CLI 來建立服務帳戶/IAM 角色時，CLI `eksctl`會自動執行此操作。如需詳細資訊，請參閱 https://eksctl.io/usage/iamserviceaccounts/：//。

直接使用 IAM 時，這會將條件新增至角色的信任政策，該政策使用條件來確保`:sub`宣告是您預期的命名空間和服務帳戶。例如，在我們擁有的 IRSA 字符具有「system：serviceaccount：default：s3-read-only」的子宣告之前。這是 `default` 命名空間，而服務帳戶是 `s3-read-only`。您可以使用類似下列的條件，以確保只有您叢集中指定命名空間中的服務帳戶可以擔任該角色：

```
  "Condition": {
```

```
      "StringEquals": {
          "oidc.eks.us-west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128:aud":
"sts.amazonaws.com",
          "oidc.eks.us-west-2.amazonaws.com/id/D43CF17C27A865933144EA99A26FB128:sub":
"system:serviceaccount:default:s3-read-only"
      }
  }
```

每個應用程式使用一個 IAM 角色

使用 IRSA 和 EKS Pod Identity 時，最佳實務是為每個應用程式提供自己的 IAM 角色。這可讓您改善隔離，因為您可以修改一個應用程式而不影響另一個應用程式，並只授予應用程式所需的許可，以套用最低權限的主體。

搭配 EKS Pod Identity 使用 ABAC 時，您可以在多個服務帳戶中使用常見的 IAM 角色，並依賴其工作階段屬性進行存取控制。這在大規模操作時特別有用，因為 ABAC 可讓您使用較少的 IAM 角色來操作 。

當您的應用程式需要存取 IMDS 時，請使用 IMDSv2，並將 EC2 執行個體的跳轉限制提高為 2

IMDSv2 要求您使用 PUT 請求來取得工作階段字符。初始 PUT 請求必須包含工作階段字符的 TTL。較新版本的 AWS SDKs會自動處理此問題和該權杖的續約。也請務必注意，EC2 執行個體的預設跳轉限制刻意設定為 1，以防止 IP 轉送。因此，請求在 EC2 執行個體上執行的工作階段字符的 Pod 最終可能會逾時，並回復為使用 IMDSv1 資料流程。EKS 透過啟用 v1 和 v2，並將 eksctl 或使用官方 CloudFormation 範本佈建的節點上的跳轉限制變更為 2，新增支援 IMDSv2。

停用服務帳戶字符的自動掛載

如果您的應用程式不需要呼叫 Kubernetes API，請在應用程式的 PodSpec `false`中將 `automountServiceAccountToken` 屬性設定為 ，或修補每個命名空間中的預設服務帳戶，使其不再自動掛載至 Pod。例如：

```
kubectl patch serviceaccount default -p $'automountServiceAccountToken: false'
```

為每個應用程式使用專用服務帳戶

每個應用程式都應有自己的專用服務帳戶。這適用於 Kubernetes API 的服務帳戶，以及 IRSA 和 EKS Pod 身分。

> **Important**
>
> 如果您在使用 IRSA 時採用藍/綠方法來叢集升級，而不是執行就地叢集升級，則需要使用新叢集的 OIDC 端點更新每個 IRSA IAM 角色的信任政策。藍/綠叢集升級可讓您建立與舊叢集一起執行較新 Kubernetes 版本的叢集，並使用負載平衡器或服務網格，將流量從舊叢集上執行的服務無縫地轉移到新叢集。搭配 EKS Pod Identity 使用藍/綠叢集升級時，您可以在新叢集的 IAM 角色和服務帳戶之間建立 Pod 身分關聯。如果您有`sourceArn`條件，請更新 IAM 角色信任政策。

以非根使用者身分執行應用程式

根據預設，容器會以根身分執行。雖然這允許他們讀取 Web 身分字符檔案，但將容器作為根執行不會被視為最佳實務。或者，請考慮將 `spec.securityContext.runAsUser` 屬性新增至 PodSpec。的值`runAsUser`是任意值。

在下列範例中，Pod 中的所有程序都會在 `runAsUser` 欄位中指定的使用者 ID 下執行。

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
```

當您以非根使用者身分執行容器時，它會防止容器讀取 IRSA 服務帳戶權杖，因為權杖預設為指派 0600 【根】 許可。如果您更新容器的 securityContext 以包含 fsgroup=65534 【無】 ，則會允許容器讀取字符。

```
spec:
  securityContext:
    fsGroup: 65534
```

在 Kubernetes 1.19 及更高版本中，不再需要此變更，應用程式可以讀取 IRSA 服務帳戶字符，而無需將其新增至沒有人群組。

授予最低權限的應用程式存取權

Action Hero 是一種公用程式，您可以與應用程式一起執行，以識別應用程式正常運作所需的 AWS API 呼叫和對應的 IAM 許可。它類似於 IAM Access Advisor，因為它可協助您逐步限制指派給應用程式的 IAM 角色範圍。如需詳細資訊，請參閱授予最低權限存取 AWS 資源的文件。

考慮在與 IRSA 和 Pod 身分搭配使用的 IAM 角色上設定許可界限。您可以使用許可界限來確保 IRSA 或 Pod 身分所使用的角色不得超過許可的最大層級。如需使用範例許可界限政策開始使用許可界限的範例指南，請參閱此 github 儲存庫。

檢閱和撤銷對 EKS 叢集不必要的匿名存取

理想情況下，所有 API 動作都應停用匿名存取。匿名存取是透過為 Kubernetes 內建使用者系統建立 RoleBinding 或 ClusterRoleBinding 來授予：匿名。您可以使用 rbac-lookup 工具來識別系統：匿名使用者在您的叢集上擁有的許可：

```
./rbac-lookup | grep -P 'system:(anonymous)|(unauthenticated)'
system:anonymous               cluster-wide        ClusterRole/system:discovery
system:unauthenticated         cluster-wide        ClusterRole/system:discovery
system:unauthenticated         cluster-wide        ClusterRole/system:public-info-
viewer
```

system：public-info-viewer 以外的任何角色或 ClusterRole 都不應繫結至 system：anonymous user or system：unuthenticated group。

在特定 APIs上啟用匿名存取可能有某些合法原因。如果您的叢集發生這種情況，請確保匿名使用者只能存取這些特定 APIs而且在沒有身分驗證的情況下公開這些 APIs 不會讓您的叢集容易受到攻擊。

在 Kubernetes/EKS 1.14 版之前，System：未驗證的群組預設與 system：discovery 和 system：basic-user ClusterRoles 相關聯。請注意，即使您已將叢集更新至 1.14 版或更新版本，叢集上仍可能啟用這些許可，因為叢集更新不會撤銷這些許可。若要檢查哪些 ClusterRoles 具有 "system：unauthenticated"，Systems：public-info-viewer 除外，您可以執行下列命令 （需要 jq util)：

```
kubectl get ClusterRoleBinding -o json | jq -r '.items[] | select(.subjects[]?.name
 =="system:unauthenticated") | select(.metadata.name != "system:public-info-viewer")
 | .metadata.name'
```

除了「system：public-info-viewer」之外，所有角色都可以使用下列方式移除「system：unauthenticated」：

```
kubectl get ClusterRoleBinding -o json | jq -r '.items[] | select(.subjects[]?.name
 =="system:unauthenticated") | select(.metadata.name != "system:public-info-viewer") |
 del(.subjects[] | select(.name =="system:unauthenticated"))' | kubectl apply -f -
```

或者，您也可以透過 kubectl 描述和 kubectl 編輯來手動檢查和移除它。若要檢查未驗證的 群組是否具有叢集上的 system：discovery 許可，請執行下列命令：

```
kubectl describe clusterrolebindings system:discovery

Name:         system:discovery
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
Role:
  Kind:  ClusterRole
  Name:  system:discovery
Subjects:
  Kind   Name                    Namespace
  ----   ----                    ---------
  Group  system:authenticated
  Group  system:unauthenticated
```

若要檢查未驗證的 群組是否具有叢集上的 system：basic-user 許可，請執行下列命令：

```
kubectl describe clusterrolebindings system:basic-user

Name:         system:basic-user
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
Role:
  Kind:  ClusterRole
  Name:  system:basic-user
Subjects:
  Kind   Name                    Namespace
  ----   ----                    ---------
  Group  system:authenticated
  Group  system:unauthenticated
```

如果 system：未驗證群組繫結至叢集上的 system：discovery 和/或 system：basic-user ClusterRoles，您應該取消這些角色與 system：未驗證群組的關聯。使用下列命令編輯 system：discovery ClusterRoleBinding：

```
kubectl edit clusterrolebindings system:discovery
```

上述命令會在編輯器中開啟 system：discovery ClusterRoleBinding 的目前定義，如下所示：

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will
 be
# reopened with the relevant failures.
#
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2021-06-17T20:50:49Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: system:discovery
  resourceVersion: "24502985"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindings/system%3Adiscovery
  uid: b7936268-5043-431a-a0e1-171a423abeb6
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:discovery
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:authenticated
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:unauthenticated
```

從上述編輯器畫面中的「主旨」區段中刪除 system：unuthenticated 群組的項目。

針對 system：basic-user ClusterRoleBinding 重複相同的步驟。

將 AWS 開發套件工作階段與 IRSA 重複使用

當您使用 IRSA 時，使用 AWS 開發套件寫入的應用程式會使用交付至 Pod 的字符來呼叫 `sts:AssumeRoleWithWebIdentity`來產生臨時 AWS 登入資料。這與其他 AWS 運算服務不同，其中運算服務會將臨時 AWS 登入資料直接交付至 AWS 運算資源，例如 lambda 函數。這表示每次初始化 AWS 開發套件工作階段時，`AssumeRoleWithWebIdentity`都會呼叫 的 AWS STS。如果您的應用程式快速擴展並初始化許多 AWS 開發套件工作階段，您可能會遇到來自 AWS STS 的限流，因為您的程式碼會對 進行許多呼叫`AssumeRoleWithWebIdentity`。

為了避免這種情況，我們建議您在應用程式中重複使用 AWS 開發套件工作階段，這樣`AssumeRoleWithWebIdentity`就不會對 進行不必要的呼叫。

在下列範例程式碼中，使用 boto3 python SDK 建立工作階段，該相同工作階段用於建立用戶端並與 Amazon S3 和 Amazon SQS 互動。 `AssumeRoleWithWebIdentity`只會呼叫一次，AWS 開發套件會在登入資料自動過期`my_session`時重新整理 的登入資料。

```
import boto3

= Create your own session

my_session = boto3.session.Session()

= Now we can create low-level clients from our session

sqs = my_session.client(``sqs``) s3 = my_session.client(``s3``)

s3response = s3.list_buckets() sqsresponse = sqs.list_queues()

#print the response from the S3 and SQS APIs print("`s3 response:`")
print(s3response) print("`—`") print("`sqs response:`")
print(sqsresponse)
```

如果您要將應用程式從另一個 AWS 運算服務遷移EC2到具有 IRSA 的 EKS，這是特別重要的詳細資訊。在其他初始化 AWS 開發套件工作階段的運算服務上，除非您指示，否則不會呼叫 AWS STS。

替代方法

雖然 IRSA 和 EKS Pod 身分是將 AWS 身分指派給 Pod 的偏好方式，但它們會要求您在應用程式中包含最新版本的 AWS SDKs。如需目前支援 IRSA 的SDKs完整清單，請參閱 https://docs.aws.amazon.com/eks/latest/userguide/iam-roles-for-service-accounts-minimum-sdk.html：// EKS Pod 身分，請參閱 https://docs.aws.amazon.com/eks/latest/userguide/pod-id-minimum-

sdk.html：//。如果您有應用程式無法立即使用相容的 SDK 更新，有數個社群建置的解決方案可用來將 IAM 角色指派給 Kubernetes Pod，包括 kube2iam 和 kiam。雖然 AWS 不認可、寬容或支援使用這些解決方案，但社群經常使用它們來達到與 IRSA 和 EKS Pod 身分類似的結果。

如果您需要使用這些非 aws 提供的解決方案之一，請進行盡職調查，並確保您了解這樣做的安全影響。

工具與資源

- Amazon EKS 安全浸入研討會 - Identity and Access Management
- Terraform EKS 藍圖模式 - 全私有 Amazon EKS 叢集
- Terraform EKS 藍圖模式 - 適用於 Amazon EKS 叢集的 IAM Identity Center 單一登入
- Terraform EKS 藍圖模式 - Okta Amazon EKS 叢集單一登入
- audit2rbac
- rbac.dev Kubernetes RBAC 的其他資源清單，包括部落格和工具
- Action Hero
- kube2iam
- kiam

Pod 安全性

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

Pod 規格包含各種不同的屬性，可強化或削弱您的整體安全狀態。身為 Kubernetes 從業人員，您的首要考量應該是防止容器中執行的程序逸出容器執行時間的隔離界限，並取得基礎主機的存取權。

Linux 功能

根據預設，在容器內執行的程序會在 【Linux】 根使用者的內容下執行。雖然容器內根的動作部分受限於容器執行時間指派給容器的一組 Linux 功能，但這些預設權限可能允許攻擊者提升其權限和/或存取繫結至主機的敏感資訊，包括 Secrets 和 ConfigMaps。以下是指派給容器的預設功能清單。如需每個功能的詳細資訊，請參閱 https：//http://man7.org/linux/man-pages/man7/capabilities.7.html。

`CAP_AUDIT_WRITE, CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_FOWNER, CAP_FSETID, CAP_KILL, CAP_MKNOD, CAP_NET_BIND_SERVICE, CAP_NET_RAW, CAP_SETGID, CAP_SETUID, CAP_SETFCAP, CAP_SETPCAP, CAP_SYS_CHROOT`

Example

EC2 和 Fargate Pod 預設會獲指派上述功能。此外，Linux 功能只能從 Fargate Pod 中刪除。

以特殊權限執行的 Pod，會繼承與主機根相關聯的所有 Linux 功能。如果可能，應該避免這種情況。

節點授權

所有 Kubernetes 工作者節點都使用稱為節點授權的授權模式。節點授權會授權來自 kubelet 的所有 API 請求，並允許節點執行下列動作：

讀取操作：

- services
- 端點
- 節點
- Pod
- 與綁定至 kubelet 節點的 Pod 相關的秘密、組態映射、持久性磁碟區宣告和持久性磁碟區

寫入操作：

- 節點和節點狀態（啟用`NodeRestriction`許可外掛程式以限制 kubelet 修改自己的節點）
- Pod 和 Pod 狀態（啟用`NodeRestriction`許可外掛程式以限制 kubelet 修改綁定至其本身的 Pod)
- events

驗證相關操作：

- TLS 引導的 CertificateSigningRequest (CSR) API 讀取/寫入存取權
- 為委派的身分驗證/授權檢查建立 TokenReview 和 SubjectAccessReview 的能力

EKS 使用節點限制許可控制器，僅允許節點修改繫結至節點的一組有限節點屬性和 Pod 物件。不過，管理 以存取主機的攻擊者仍然可以從 Kubernetes API 收集環境的敏感資訊，讓他們可以在叢集內橫向移動。

Pod 安全解決方案

Pod 安全政策 (PSP)

在過去，Pod 安全政策 (PSP) 資源用於指定一組 Pod 在建立之前必須符合的要求。自 Kubernetes 1.21 版起，PSP 已棄用。它們排定在 Kubernetes 1.25 版中移除。

> **Important**
>
> PSPs Kubernetes 1.21 版中已棄用。您將擁有直到 1.25 版或大約 2 年才能轉換為替代版本。本文件說明此棄用的動機。

遷移至新的 Pod 安全解決方案

由於 PSPs已從 Kubernetes v1.25 移除，叢集管理員和運算子必須取代這些安全控制。兩種解決方案可以滿足此需求：

- Kubernetes 生態系統中的Policy-as-code(PAC) 解決方案
- Kubernetes Pod 安全標準 (PSS)

PAC 和 PSS 解決方案都可以與 PSP 共存；它們可以在移除 PSP 之前用於叢集。從 PSP 遷移時，這可簡化採用。考慮從 PSP 遷移至 PSS 時，請參閱本文件。

Kyverno 是下列概述的 PAC 解決方案之一，在從 PSPs 遷移至其解決方案時，有部落格文章中概述的特定指引，包括類似政策、功能比較和遷移程序。有關遷移至 Kyverno 有關 Pod 安全許可 (PSA) 的其他資訊和指導已發佈於此處的 AWS 部落格。

Policy-as-code(PAC)

Policy-as-code(PAC) 解決方案提供護欄，透過規定和自動化控制來引導叢集使用者，並防止不必要的行為。PAC 使用 Kubernetes 動態許可控制器透過 Webhook 呼叫攔截 Kubernetes API 伺服器請求流程，並根據撰寫並儲存為程式碼的政策來變更和驗證請求承載。變更和驗證會在 API 伺服器請求導致叢集變更之前發生。PAC 解決方案使用政策來比對 API 伺服器請求承載，並根據分類和值採取行動。

有多種開放原始碼 PAC 解決方案可供 Kubernetes 使用。這些解決方案不屬於 Kubernetes 專案；它們來自 Kubernetes 生態系統。以下列出一些 PAC 解決方案。

- OPA/Gatekeeper

- 開啟政策代理程式 (OPA)
- 基佛諾
- Kubewarden
- jsPolicy

如需 PAC 解決方案以及如何協助您選擇適合您需求的解決方案的詳細資訊，請參閱以下連結。

- Kubernetes 的政策型對策 – 第 1 部分
- Kubernetes 的政策型對策 – 第 2 部分

Pod 安全標準 (PSS) 和 Pod 安全許可 (PSA)

為了回應 PSP 棄用和持續需要out-of-the-box控制 Pod 安全，使用內建的 Kubernetes 解決方案，Kubernetes Auth Special Interest Group 建立了 Pod 安全標準 (PSS) 和 Pod 安全許可 (PSA)。PSA 工作包含許可控制器 Webhook 專案，可實作 PSS 中定義的控制項。此許可控制器方法類似於 PAC 解決方案中使用的 。

根據 Kubernetes 文件，PSS「`定義三個不同的政策，以廣泛涵蓋安全範圍。這些政策是累積的，範圍從高度寬鬆到高度限制。`"

這些政策的定義如下：

- 特殊權限：不受限制（不安全）政策，提供最廣泛的許可層級。此政策允許已知的權限提升。這是缺少政策。這適用於記錄代理程式、CNIs、儲存驅動程式等應用程式，以及其他需要特殊存取權限的應用程式。
- 基準：防止已知權限提升的最小限制政策。允許預設（最低指定）Pod 組態。基準政策禁止使用 hostNetwork、hostPID、hostIPC、hostPath、hostPort、無法新增 Linux 功能，以及其他幾項限制。
- 限制：嚴格限制政策，遵循目前的 Pod 強化最佳實務。此政策繼承自基準，並新增進一步的限制，例如無法以根或根群組身分執行。限制政策可能會影響應用程式運作的能力。它們主要以執行安全關鍵應用程式為目標。

這些政策會定義 Pod 執行的設定檔，並分為三個層級的特權與 限制存取。

若要實作 PSS 定義的控制項，PSA 會以三種模式運作：

- 強制執行：違反政策會導致 Pod 遭到拒絕。

- 稽核：政策違規將觸發將稽核註釋新增至稽核日誌中記錄的事件，但在其他情況下是允許的。
- 警告：違反政策將觸發面向使用者的警告，但在其他情況下是允許的。

這些模式和設定檔（限制）層級是在 Kubernetes 命名空間層級使用標籤設定，如下列範例所示。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/enforce: restricted
```

獨立使用時，這些操作模式會有不同的回應，進而產生不同的使用者體驗。如果個別 PodSpecs 違反設定的限制層級，強制執行模式將阻止建立 Pod。 podSpecs 不過，在此模式中，即使其中的 podSpec 違反套用的 PSS，也不會阻止建立 Pod 的非 Pod Kubernetes 物件，例如部署。在此情況下，部署將會套用，而 Pod (s) 將會無法套用。

這是一個困難的使用者體驗，因為沒有立即指出成功套用的部署物件是失敗的 Pod 建立。違規的 podSpecs 不會建立 Pod。使用 檢查部署資源`kubectl get deploy <DEPLOYMENT_NAME> -oyaml`會公開來自失敗 Pod (s) `.status.conditions`元素的訊息，如下所示。

```
...
status:
  conditions:
    - lastTransitionTime: "2022-01-20T01:02:08Z"
      lastUpdateTime: "2022-01-20T01:02:08Z"
      message: 'pods "test-688f68dc87-tw587" is forbidden: violates PodSecurity
 "restricted:latest":
        allowPrivilegeEscalation != false (container "test" must set
 securityContext.allowPrivilegeEscalation=false),
        unrestricted capabilities (container "test" must set
 securityContext.capabilities.drop=["ALL"]),
        runAsNonRoot != true (pod or container "test" must set
 securityContext.runAsNonRoot=true),
        seccompProfile (pod or container "test" must set
 securityContext.seccompProfile.type
        to "RuntimeDefault" or "Localhost")'
      reason: FailedCreate
      status: "True"
      type: ReplicaFailure
```

```
...
```

在稽核和警告模式中，Pod 限制不會防止建立和啟動違反 Pod。不過，在這些模式中，當 Pod 以及建立 Pod 的物件包含違規的 podSpecs 時，會分別觸發 API 伺服器稽核日誌事件和警告上的稽核註釋，例如 kubectl。警告kubectl訊息如下所示。

```
Warning: would violate PodSecurity "restricted:latest":
 allowPrivilegeEscalation != false (container "test" must set
 securityContext.allowPrivilegeEscalation=false), unrestricted capabilities (container
 "test" must set securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod
 or container "test" must set securityContext.runAsNonRoot=true), seccompProfile (pod
 or container "test" must set securityContext.seccompProfile.type to "RuntimeDefault"
 or "Localhost")
deployment.apps/test created
```

引進 PSS 時，PSA 稽核和警告模式非常有用，而不會對叢集操作造成負面影響。

PSA 操作模式並非互斥，可以累積方式使用。如下所示，您可以在單一命名空間中設定多個模式。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
```

在上述範例中，套用部署時會提供易於使用的警告和稽核註釋，而強制執行違規也會在 Pod 層級提供。事實上，多個 PSA 標籤可以使用不同的設定檔層級，如下所示。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/enforce: baseline
    pod-security.kubernetes.io/warn: restricted
```

在上述範例中，PSA 設定為允許建立符合基準設定檔層級的所有 Pod，然後在違反受限設定檔層級的 Pod（以及建立 Pod 的物件）上發出警告。這是一個有用的方法，可判斷從基準變更為受限設定檔時可能產生的影響。

現有 Pod

如果修改具有現有 Pod 的命名空間以使用更嚴格的 PSS 設定檔，則稽核和警告模式會產生適當的訊息；不過，強制模式不會刪除 Pod。警告訊息如下所示。

```
Warning: existing pods in namespace "policy-test" violate the new PodSecurity enforce
 level "restricted:latest"
Warning: test-688f68dc87-htm8x: allowPrivilegeEscalation != false, unrestricted
 capabilities, runAsNonRoot != true, seccompProfile
namespace/policy-test configured
```

豁免

PSA 使用豁免來排除對原本已套用的 Pod 強制執行違規。這些豁免列於下方。

- 使用者名稱：忽略具有豁免身分驗證（或模擬）使用者名稱之使用者的請求。
- RuntimeClassNames：會忽略指定豁免執行階段類別名稱的 Pod 和工作負載資源。
- 命名空間：忽略豁免命名空間中的 Pod 和工作負載資源。

這些豁免會在 PSA 許可控制器組態中靜態套用，做為 API 伺服器組態的一部分。

在驗證 Webhook 實作中，可以在 Kubernetes ConfigMap 資源中設定豁免，該資源會掛載為 pod-security-webhook 容器的磁碟區。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pod-security-webhook
  namespace: pod-security-webhook
data:
  podsecurityconfiguration.yaml: |
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration
    defaults:
      enforce: "restricted"
      enforce-version: "latest"
```

```
      audit: "restricted"
      audit-version: "latest"
      warn: "restricted"
      warn-version: "latest"
    exemptions:
      # Array of authenticated usernames to exempt.
      usernames: []
      # Array of runtime class names to exempt.
      runtimeClasses: []
      # Array of namespaces to exempt.
      namespaces: ["kube-system","policy-test1"]
```

如上述 ConfigMap YAML 所示，整個叢集的預設 PSS 層級已設定為限制所有 PSA 模式、稽核、強制執行和警告。這會影響所有命名空間，但不包括豁免的命名空間：`namespaces: ["kube-system","policy-test1"]`。此外，在如下所示的 ValidatingWebhookConfiguration 資源中，pod-security-webhook 命名空間也會從設定的 PSS 中排除。

```
...
webhooks:
  # Audit annotations will be prefixed with this name
  - name: "pod-security-webhook.kubernetes.io"
    # Fail-closed admission webhooks can present operational challenges.
    # You may want to consider using a failure policy of Ignore, but should
    # consider the security tradeoffs.
    failurePolicy: Fail
    namespaceSelector:
      # Exempt the webhook itself to avoid a circular dependency.
      matchExpressions:
        - key: kubernetes.io/metadata.name
          operator: NotIn
          values: ["pod-security-webhook"]
...
```

> **Important**
>
> Pod 安全許可在 Kubernetes 1.25 版中進入穩定狀態。如果您想要在預設啟用 Pod 安全許可功能之前使用此功能，您需要安裝動態許可控制器（變動 Webhook)。您可以在此處找到安裝和設定 Webhook 的說明。

在policy-as-code和 Pod 安全標準之間進行選擇

Pod 安全標準 (PSS) 旨在取代 Pod 安全政策 (PSP)，方法是提供內建於 Kubernetes 且不需要 Kubernetes 生態系統解決方案的解決方案。也就是說，policy-as-code(PAC) 解決方案更加靈活。

下列 Pros 和 Cons 清單旨在協助您更明智地決定 Pod 安全解決方案。

Policy-as-code（與 Pod 安全標準相比）

優點：

- 更靈活、更精細（視需要縮減至資源的屬性）
- 不僅專注於 Pod，還可以用於不同的資源和動作
- 不僅適用於命名空間層級
- 比 Pod 安全標準更成熟
- 決策可以根據 API 伺服器請求承載中的任何內容，以及現有的叢集資源和外部資料（取決於解決方案）
- 支援在驗證之前變更 API 伺服器請求（取決於解決方案）
- 可以產生互補政策和 Kubernetes 資源（取決於解決方案 - 從 Pod 政策中，Kyverno 可以自動產生高階控制器的政策，例如部署。 Kyverno 也可以使用產生規則產生額外的 Kubernetes 資源「建立新資源時或來源更新時`」。）
- 在呼叫 Kubernetes API 伺服器（依解決方案而定）之前，可用來向左移動至 CICD 管道
- 可用於實作不一定與安全相關的行為，例如最佳實務、組織標準等。
- 可用於非 Kubernetes 使用案例（取決於解決方案）
- 由於靈活性，使用者體驗可以根據使用者的需求進行調整

缺點：

- 未內建於 Kubernetes
- 更複雜的學習、設定和支援
- 政策撰寫可能需要新的skills/languages/capabilities

Pod 安全許可（與policy-as-code相比）

優點：

- 內建於 Kubernetes
- 設定更簡單
- 沒有要使用的新語言或編寫的政策
- 如果叢集預設許可層級設定為特殊權限，則可以使用命名空間標籤將命名空間選擇為 Pod 安全性設定檔。

缺點：

- 不像policy-as-code一樣靈活或精細
- 只有 3 個層級的限制
- 主要專注於 Pod

摘要

如果您目前沒有 PSP 以外的 Pod 安全解決方案，且所需的 Pod 安全狀態符合 Pod 安全標準 (PSS) 中定義的模型，則更簡單的路徑可能是採用 PSS，而不是policy-as-code解決方案。不過，如果您的 Pod 安全狀態不符合 PSS 模型，或者您設想新增超出 PSS 定義的其他控制項，則policy-as-code解決方案似乎更合適。

建議

使用多個 Pod 安全許可 (PSA) 模式，以獲得更好的使用者體驗

如前所述，PSA 強制執行模式可防止套用具有 PSS 違規的 Pod，但不會停止更高層級的控制器，例如部署。事實上，部署將成功套用，而不會指出 Pod 無法套用。雖然您可以使用 kubectl 來檢查部署物件，並從 PSA 探索失敗的 Pod 訊息，但使用者體驗可能會更好。為了改善使用者體驗，應使用多個 PSA 模式（稽核、強制執行、警告）。

```
apiVersion: v1
kind: Namespace
metadata:
  name: policy-test
  labels:
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/warn: restricted
```

在上述範例中，在定義強制執行模式的情況下，當嘗試將個別 podSpec 中具有 PSS 違規的部署資訊清單套用至 Kubernetes API 伺服器時，部署將成功套用，但 Pod 不會。此外，由於稽核和警告模式也已啟用，API 伺服器用戶端會收到警告訊息，API 伺服器稽核日誌事件也會以訊息標註。

限制可做為特殊權限執行的容器

如上所述，做為特殊權限執行的容器會繼承指派給主機根目錄的所有 Linux 功能。容器很少需要這些類型的權限才能正常運作。有多種方法可用來限制容器的許可和功能。

> **Important**
>
> Fargate 是一種啟動類型，可讓您執行「無伺服器」容器，其中 Pod 的容器會在 AWS 管理的基礎設施上執行。使用 Fargate，您無法執行特權容器或將 Pod 設定為使用 hostNetwork 或 hostPort。

請勿在容器中執行程序做為根目錄

根據預設，所有容器都會以根身分執行。如果攻擊者能夠利用應用程式中的漏洞，並取得執行中容器的 shell 存取，則這可能有問題。您可以用各種方式降低此風險。首先，從容器映像中移除 Shell。其次，將 USER 指令新增至您的 Dockerfile，或以非根使用者身分在 Pod 中執行容器。Kubernetes podSpec 包含一組欄位，位於 下`spec.securityContext`，可讓您指定要在其中執行應用程式的使用者和/或群組。這些欄位`runAsGroup`分別為 `runAsUser`和 。

若要在 Kubernetes podSpec 中強制使用 `spec.securityContext`及其相關聯的元素，可將policy-as-code或 Pod 安全標準新增至叢集。這些解決方案可讓您撰寫和/或使用可驗證傳入 Kubernetes API 伺服器請求承載的政策或設定檔，然後再保留到 等。此外，policy-as-code解決方案可以改變傳入請求，在某些情況下，會產生新的請求。

切勿在 Docker 中執行 Docker 或在容器中掛載通訊端

雖然這可讓您在 Docker 容器中建置/執行映像，但您基本上會放棄節點對容器中執行程序的完整控制。如果您需要在 Kubernetes 上建置容器映像，請改用 Kaniko、buildah 或像是 CodeBuild 的建置服務。

> **Note**
>
> 用於 CICD 處理的 Kubernetes 叢集，例如建置容器映像，應與執行更廣泛工作負載的叢集隔離。

限制使用 hostPath 或如果需要 hostPath 限制可使用的字首，並將磁碟區設定為唯讀

hostPath 是將目錄從主機直接掛載到容器的磁碟區。在極少數情況下，Pod 會需要這種類型的存取，但如果他們這麼做，您需要注意風險。根據預設，以根身分執行的 Pod 將具有 hostPath 公開之檔案系統的寫入存取權。這可能會允許攻擊者修改 kubelet 設定、建立未由 hostPath 直接公開的目錄或檔案的符號連結，例如 /etc/shadow、安裝 ssh 金鑰、掛載到主機的讀取秘密，以及其他惡意物件。若要降低 hostPath 的風險，請將 設定為 `spec.containers.volumeMounts` `readOnly`，例如：

```
volumeMounts:
- name: hostPath-volume
    readOnly: true
    mountPath: /host-path
```

您也應該使用policy-as-code解決方案來限制可供磁碟`hostPath`區使用的目錄，或完全防止`hostPath`使用。您可以使用 Pod 安全標準基準或受限制政策來防止使用 `hostPath`。

如需特殊權限提升危險的進一步資訊，請參閱 Seth Art 的部落格錯誤 Pod：Kubernetes Pod 特殊權限提升。

為每個容器設定請求和限制，以避免資源爭用和 DoS 攻擊

沒有請求或限制的 Pod 理論上可以使用主機上所有可用的資源。當其他 Pod 排程到節點時，節點可能會遇到 CPU 或記憶體壓力，這可能會導致 Kubelet 從節點終止或移出 Pod。雖然您無法防止這種情況一起發生，但設定請求和限制將有助於最大限度地減少資源爭用，並降低因寫入不良的應用程式消耗過多資源的風險。

podSpec 可讓您指定 CPU 和記憶體的請求和限制。CPU 被視為可壓縮資源，因為它可能會過度訂閱。記憶體無法壓縮，即 無法在多個容器之間共用。

當您指定 CPU 或記憶體的請求時，基本上就是指定容器保證取得的記憶體量。Kubernetes 會彙總 Pod 中所有容器的請求，以決定要排程 Pod 的節點。如果容器超過請求的記憶體數量，如果節點上有記憶體壓力，則可能會終止。

限制是允許容器使用並直接對應至為容器建立之 cgroup `memory.limit_in_bytes`值的 CPU 和記憶體資源數量上限。超過記憶體限制的容器將會終止 OOM。如果容器超過其 CPU 限制，則會對其進行調節。

> **Note**
>
> 使用容器時`resources.limits`，強烈建議容器資源使用量 (a.k.a. Resource Footprints) 根據負載測試以資料驅動且準確。如果沒有準確且受信任的資源使用

量，resources.limits可以填充容器。例如， resources.limits.memory可能填充比可觀測的最大值高 20-30%，以考慮潛在的記憶體資源限制不正確。

Kubernetes 使用三種服務品質 (QoS) 類別來排定節點上執行工作負載的優先順序。其中包含：

- 保證
- 爆量
- 最佳努力

如果未設定限制和請求，Pod 會設定為盡力 （最低優先順序）。當記憶體不足時，最努力的 Pod 是第一個被終止的 Pod。如果 Pod 內的所有容器都設定了限制，或者如果請求和限制設定為相同的值且不等於 0，則 Pod 會設定為保證 （最高優先順序）。除非保證的 Pod 超過其設定的記憶體限制，否則不會被終止。如果限制和請求設定了不同的值，不等於 0，或 Pod 中的一個容器設定了限制，而其他容器未設定或設定了不同資源的限制，則 Pod 會設定為爆量 （中等優先順序）。這些 Pod 有一些資源保證，但一旦超過其請求的記憶體，就會遭到刪除。

Important

請求不會影響容器 cgroup memory_limit_in_bytes的值；cgroup 限制會設定為主機上可用的記憶體數量。不過，如果設定過低的請求值，可能會導致 Pod 在節點經歷記憶體壓力時成為 kubelet 終止的目標。

類別	Priority	條件	刪除條件
保證	最高	limit = 請求 ！= 0	僅超過記憶體限制
爆量	中型	限制 ！= 請求 ！= 0	如果超過請求記憶體，則可以終止
盡最大努力	最低	限制和請求未設定	記憶體不足時首先被終止

如需資源 QoS 的其他資訊，請參閱 Kubernetes 文件。

您可以在命名空間上設定資源配額或建立限制範圍，強制使用請求和限制。資源配額可讓您指定配置給命名空間的資源總量，例如 CPU 和 RAM。套用到命名空間時，它會強制您指定部署到該命名空間中所有容器的請求和限制。相反地，限制範圍可讓您更精細地控制資源的配置。透過限制範圍，您可以為每個 Pod 或命名空間內每個容器的 CPU 和記憶體資源最小/最大。如果未提供任何請求/限制值，您也可以使用這些值來設定預設請求/限制值。

Policy-as-code解決方案可用於強制執行請求和限制。 或 甚至在建立命名空間時建立資源配額和限制範圍。

不允許特殊權限呈報

特殊權限提升允許程序變更其執行所在的安全內容。Sudo 是很好的範例，因為 是具有 SUID 或 SGID 位元的二進位檔。特殊權限呈報基本上是使用者執行具有其他使用者或群組許可的檔案的一種方式。您可以在 `securityContext.allowPrivilegeEscalation`中實作設定為 `allowPrivilegeEscalationfalse`或 policy-as-code變更政策，以防止容器使用特殊權限升級`podSpec`。如果偵測到不正確的設定，Policy-as-code政策也可以用來防止 API 伺服器請求成功。Pod 安全標準也可以用來防止 Pod 使用權限提升。

停用 ServiceAccount 字符掛載

對於不需要存取 Kubernetes API 的 Pod，您可以在 Pod 規格上停用 ServiceAccount 字符的自動掛載，或停用使用特定 ServiceAccount 的所有 Pod。

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-no-automount
spec:
  automountServiceAccountToken: false
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-no-automount
automountServiceAccountToken: false
```

停用服務探索

對於不需要查詢或呼叫叢集內服務的 Pod，您可以減少提供給 Pod 的資訊量。您可以將 Pod 的 DNS 政策設定為不使用 CoreDNS，且不得將 Pod 命名空間中的服務公開為環境變數。如需服務連結的詳細資訊，請參閱環境變數上的 Kubernetes 文件。Pod 的 DNS 政策預設值為「ClusterFirst」，其使用叢集內 DNS，而非預設值「Default」則使用基礎節點的 DNS 解析。如需詳細資訊，請參閱 Pod DNS 政策上的 Kubernetes 文件。

Example

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-no-service-info
spec:
    dnsPolicy: Default # "Default" is not the true default value
    enableServiceLinks: false
```

使用唯讀根檔案系統設定映像

使用唯讀根檔案系統設定映像，可防止攻擊者覆寫應用程式使用的檔案系統上的二進位檔案。如果您的應用程式必須寫入檔案系統，請考慮寫入暫存目錄或連接並掛載磁碟區。您可以透過設定 Pod 的 SecurityContext 來強制執行此操作，如下所示：

```
...
securityContext:
  readOnlyRootFilesystem: true
...
```

Policy-as-code和 Pod 安全標準可用來強制執行此行為。

Example

根據 Kubernetes 中的 Windows 容器，`securityContext.readOnlyRootFilesystem`無法在 Windows 上執行`true`的容器設定為 ，因為需要寫入存取權，登錄檔和系統程序才能在容器內執行。

工具和資源

- Amazon EKS 安全沉浸研討會 - Pod 安全
- open-policy-agent/gatekeeper-library：OPA Gatekeeper 政策程式庫 OPA/Gatekeeper 政策程式庫，可用來取代 PSPs。

- Kyverno 政策程式庫
- EKS 的常見 OPA 和 Kyverno 政策集合。
- 政策型對策：第 1 部分
- 政策型對策：第 2 部分
- Pod 安全政策 Migrator 將 PSPs 轉換為 OPA/Gatekeeper、KubeWarden 或 Kyverno 政策的工具
- NeuVector by SUSE 開放原始碼、零信任容器安全平台，提供程序和檔案系統政策以及許可控制規則。

租戶隔離

當我們考慮多租戶時，我們通常想要隔離使用者或應用程式與在共用基礎設施上執行的其他使用者或應用程式。

Kubernetes 是單一租用戶協調程式，即控制平面的 單一執行個體在叢集中的所有租用戶之間共用。不過，您可以使用各種 Kubernetes 物件來建立多租用戶的相似性。例如，可以實作命名空間和角色型存取控制 (RBAC)，以邏輯方式將租用戶彼此隔離。同樣地，配額和限制範圍可用來控制每個租用戶可以使用的叢集資源數量。不過，叢集是唯一提供強大安全界限的建構。這是因為管理 以取得叢集內主機存取權的攻擊者可以擷取掛載在該主機上的所有秘密、ConfigMaps 和磁碟區。他們也可以模擬 Kubelet，讓他們可以操作節點的屬性和/或在叢集內橫向移動。

以下各節將說明如何實作租用戶隔離，同時降低使用 Kubernetes 等單一租用戶協調器的風險。

軟式多租用戶

透過軟式多租用戶，您可以使用原生 Kubernetes 建構模組，例如 命名空間、角色和角色繫結，以及網路政策，在租用戶之間建立邏輯分隔。例如，RBAC 可以防止租戶存取或操作彼此的資源。配額和限制範圍控制每個租用戶可以使用的叢集資源量，而網路政策可協助防止部署到不同命名空間的應用程式彼此通訊。

不過，這些控制項都無法防止來自不同租用戶的 Pod 共用節點。如果需要更強的隔離，您可以使用節點選擇器、反親和性規則和/或污點和容錯，強制將來自不同租用戶的 Pod 排程到不同的節點；通常稱為唯一租用戶節點。這可能會在具有許多租用戶的環境中變得相當複雜且成本高昂。

> **Important**
>
> 使用命名空間實作的軟式多租用不允許您為租戶提供篩選後的命名空間清單，因為命名空間是全域範圍類型。如果租用戶能夠檢視特定命名空間，則可以檢視叢集中的所有命名空間。

> **Warning**
>
> 使用soft-multi-tenancy時，租用戶保留查詢 CoreDNS 的功能，以查詢預設在叢集內執行的所有服務。攻擊者可以透過 `..svc.cluster.local`從叢集中的任何 Pod 執行 dig SRV 來利用這一點。如果您需要限制存取叢集內所執行服務的 DNS 記錄，請考慮使用 CoreDNS 的防火牆或政策外掛程式。如需詳細資訊，請參閱 https：//https://github.com/coredns/policy#kubernetes-metadata-multi-tenancy-policy。

Kiosk 是一種開放原始碼專案，可協助實作軟式多租用戶。它實作為一系列 CRDs和控制器，提供下列功能：

- 帳戶和帳戶 使用者在共用的 Kubernetes 叢集中分隔租用戶
- 帳戶使用者的自助式命名空間佈建
- 帳戶限制，以確保共享叢集時的服務品質和公平性
- 用於安全租戶隔離和自助式命名空間初始化的命名空間範本

Loft 是來自 Kiosk 和 DevSpace 維護者的商務產品，新增了下列功能：

- 授予不同叢集中空間存取權的多叢集存取權
- 休眠模式會在閒置期間縮減空間中的部署
- 使用 GitHub 等 OIDC 身分驗證供應商的單一登入

軟式多租用戶可以解決三個主要使用案例。

企業設定

第一個是企業設定，其中「租用戶」是半信任的，他們是員工、承包商或組織以其他方式授權。每個租戶通常會符合行政部門，例如部門或團隊。

在此類型的設定中，叢集管理員通常會負責建立命名空間和管理政策。他們也可能實作委派的管理模型，讓某些個人能夠監督命名空間，讓他們能夠對部署、服務、Pod、任務等非政策相關物件執行 CRUD 操作。

在此設定中，容器執行時間提供的隔離可能是可接受的，或者可能需要增強額外的 Pod 安全控制。如果需要更嚴格的隔離，也可能需要限制不同命名空間中服務之間的通訊。

Kubernetes as a Service

相反地，軟式多租用戶可用於您想要提供 Kubernetes 即服務 (KaaS) 的設定。使用 KaaS，您的應用程式會託管在共用叢集中，以及一組提供一組 PaaS 服務的控制器和 CRDs。租用戶會直接與 Kubernetes API 伺服器互動，並允許 對非政策物件執行 CRUD 操作。中也有自助服務的 元素，讓租戶可以建立和管理自己的命名空間。在此類型的環境中，租用戶會假設執行不受信任的程式碼。

若要隔離此類環境中的租戶，您可能需要實作嚴格的網路政策以及 Pod 沙盒。沙盒是您在 Firecracker 等微型 VM 或使用者空間核心中執行 Pod 容器的地方。今天，您可以使用 EKS Fargate 建立沙盒化 Pod。

軟體即服務 (SaaS)

軟式多租用戶的最終使用案例為Software-as-a-Service(SaaS) 設定。在此環境中，每個租用戶都會與叢集內執行之應用程式的特定執行個體相關聯。每個執行個體通常都有自己的資料，並使用通常與 Kubernetes RBAC 無關的個別存取控制。

與其他使用案例不同，SaaS 設定中的租用戶不會直接與 Kubernetes API 連接。反之，SaaS 應用程式負責與 Kubernetes API 互動，以建立支援每個租用戶的必要物件。

Kubernetes 建構

在每個執行個體中，下列建構用於將租用戶彼此隔離：

命名空間

命名空間是實作軟式多租用戶的基礎。它們可讓您將叢集分割為邏輯分割區。實作多租用戶所需的配額、網路政策、服務帳戶和其他物件，範圍限定於命名空間。

網路政策

根據預設，Kubernetes 叢集中的所有 Pod 都可以彼此通訊。您可以使用網路政策來變更此行為。

網路政策會使用標籤或 IP 地址範圍來限制 Pod 之間的通訊。在需要租用戶之間嚴格網路隔離的多租用戶環境中，我們建議從拒絕 Pod 之間通訊的預設規則開始，以及允許所有 Pod 查詢 DNS 伺服器以進行名稱解析的另一個規則開始。有了該功能，您就可以開始新增更多允許命名空間內通訊的寬鬆規則。這可以視需要進一步改進。

> **Note**
>
> Amazon VPC CNI 現在支援 Kubernetes 網路政策，以建立可隔離敏感工作負載的政策，並在 AWS 上執行 Kubernetes 時保護這些工作負載免於未經授權的存取。這表示您可以在 Amazon

EKS 叢集中使用網路政策 API 的所有功能。此層級的精細控制可讓您實作最低權限原則，以確保僅允許授權的 Pod 彼此通訊。

> **Important**
>
> 網路政策是必要的，但不夠。網路政策的強制執行需要政策引擎，例如 Calico 或 Cilium。

角色型存取控制 (RBAC)

角色和角色繫結是用於在 Kubernetes 中強制執行角色型存取控制 (RBAC) 的 Kubernetes 物件。角色包含可針對叢集中的物件執行的動作清單。角色繫結指定角色適用的個人或群組。在企業和 KaaS 設定中，RBAC 可用來允許由選取的群組或個人管理物件。

配額

配額用於定義叢集中託管工作負載的限制。透過配額，您可以指定 Pod 可以使用的 CPU 和記憶體數量上限，也可以限制叢集或命名空間中可配置的資源數量。限制範圍可讓您宣告每個限制的最小值、最大值和預設值。

過度承諾共用叢集中的資源通常很有幫助，因為它可讓您將資源最大化。不過，對叢集的無限制存取可能會導致資源匱乏，這可能會導致效能降低和應用程式可用性的損失。如果 Pod 的請求設定過低，且實際資源使用率超過節點的容量，則節點會開始遇到 CPU 或記憶體壓力。發生這種情況時，可能會重新啟動和/或從節點移出 Pod。

為了防止這種情況發生，您應該計劃在多租戶環境中的命名空間上強加配額，以強制租戶在叢集上排程其 Pod 時指定請求和限制。它還將限制 Pod 可以使用的資源量，以減少潛在的拒絕服務。

您也可以使用配額來分配叢集的資源，以符合租戶的支出。這在 KaaS 案例中特別有用。

Pod 優先順序和先佔

當您想要提供比其他 Pod 更重要的 Pod 時，Pod 優先順序和先佔可能很有用。例如，透過 Pod 優先順序，您可以設定客戶 A 的 Pod 以高於客戶 B 的優先順序執行。當可用容量不足時，排程器會從客戶 B 移出較低優先順序的 Pod，以容納客戶 A 的較高優先順序 Pod。這在 SaaS 環境中特別方便，因為客戶願意支付溢價會獲得較高的優先順序。

> **Important**
>
> Pod 優先順序可能會對優先順序較低的其他 Pod 產生不必要的效果。例如，雖然受害者的 Pod 已正常終止，但不保證 PodDisruptionBudget，這可能會破壞依賴一定數量 Pod 的優先順序較低的應用程式，請參閱先佔限制。

緩解控制項

您身為多租用戶環境管理員的主要考量是阻止攻擊者存取基礎主機。應考慮以下控制項以降低此風險：

容器的沙盒化執行環境

沙盒是一種技術，每個容器都會在自己的隔離虛擬機器中執行。執行 Pod 沙盒的技術包括 Firecracker 和 Weave 的 Firekube。

如需讓 Firecracker 成為 EKS 支援執行時間的詳細資訊，請參閱 https：//https://threadreaderapp.com/thread/1238496944684597248.html。

開啟政策代理程式 (OPA) 和 Gatekeeper

Gatekeeper 是 Kubernetes 許可控制器，可強制執行使用 OPA 建立的政策。使用 OPA，您可以建立政策，從不同執行個體上的租用戶執行 Pod，或以高於其他租用戶的優先順序執行 Pod。您可以在此專案的 GitHub 儲存庫中找到常見 OPA 政策的集合。

還有 CoreDNS 的實驗性 OPA 外掛程式，可讓您使用 OPA 來篩選/控制 CoreDNS 傳回的記錄。

基佛諾

Kyverno 是 Kubernetes 原生政策引擎，可使用政策做為 Kubernetes 資源來驗證、變更和產生組態。Kyverno 使用 Kustomize 樣式的浮水印進行驗證、支援 JSON 修補程式和變動的策略合併修補程式，並且可以根據彈性觸發在命名空間之間複製資源。

您可以使用 Kyverno 隔離命名空間、強制執行 Pod 安全性和其他最佳實務，以及產生預設組態，例如網路政策。此專案的 GitHub 儲存庫中包含數個範例。Kyverno 網站上的政策程式庫中包含許多其他項目。

將租戶工作負載隔離到特定節點

限制在特定節點上執行的租戶工作負載，可用來增加軟式多租戶模型中的隔離。透過此方法，租戶特定的工作負載只會在為個別租戶佈建的節點上執行。為了實現此隔離，原生 Kubernetes 屬性（節點親和

性、污點和容錯）用於鎖定特定節點以進行 Pod 排程，並防止來自其他租用戶的 Pod 排程在租用戶特定的節點上。

第 1 部分 - 節點親和性

Kubernetes 節點親和性用於根據節點標籤將節點設為目標以進行排程。使用節點親和性規則，Pod 會被吸引到符合選取器詞彙的特定節點。在下列 Pod 規格中，`requiredDuringSchedulingIgnoredDuringExecution`節點親和性會套用至個別的 Pod。結果是 Pod 將以以下列索引鍵/值標記的節點為目標：`node-restriction.kubernetes.io/tenant: tenants-x`。

```
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: node-restriction.kubernetes.io/tenant
            operator: In
            values:
            - tenants-x
...
```

使用此節點親和性時，在排程期間需要標籤，但在執行期間則不需要；如果基礎節點的標籤變更，則不會僅因為該標籤變更而移出 Pod。不過，未來的排程可能會受到影響。

> **Warning**
>
> 的標籤字首在 Kubernetes 中`node-restriction.kubernetes.io/`具有特殊意義。為 EKS 叢集啟用的 NodeRestriction `kubelet` 可防止adding/removing/updating具有此字首的標籤。攻擊者無法使用 ，`kubelet's credentials to update the node object or modify the system setup to pass these labels into `kubelet`因為 kubelet 不允許修改這些標籤。如果此字首用於所有 Pod 到節點排程，則可防止攻擊者透過修改節點標籤，將不同的工作負載集吸引到節點的情況。

Example

我們可以使用節點選擇器，而不是節點親和性。不過，節點親和性更具表達性，並允許在 Pod 排程期間考慮更多條件。如需差異和更進階排程選項的詳細資訊，請參閱此 CNCF 部落格文章中的進階 Kubernetes Pod 到節點排程。

第 2 部分 - 污點和容錯

吸引 Pod 到節點只是這個三部分方法的第一部分。若要讓此方法運作，我們必須避免 Pod 排程到未獲授權 Pod 的節點。為了避免不必要的或未經授權的 Pod，Kubernetes 會使用節點污點。污點用於在節點上放置條件，以防止 Pod 排程。下列污點使用 的鍵/值對`tenant: tenants-x`。

```
...
    taints:
      - key: tenant
        value: tenants-x
        effect: NoSchedule
...
```

鑑於上述節點 `taint`，僅允許在節點上排程可容忍污點的 Pod。若要允許將授權的 Pod 排程到節點上，個別的 Pod 規格必須包含污點`toleration`的 ，如下所示。

```
...
  tolerations:
  - effect: NoSchedule
    key: tenant
    operator: Equal
    value: tenants-x
...
```

具有上述內容的 Pod `toleration`不會停止在節點上排程，至少不會因為該特定污點而停止排程。Kubernetes 也會使用污點，在特定情況下暫時停止 Pod 排程，例如節點資源壓力。透過節點親和性、污點和容錯，我們可以有效地將所需的 Pod 吸引到特定節點，並排斥不需要的 Pod。

> **Important**
>
> 某些 Kubernetes Pod 需要在所有節點上執行。這些 Pod 的範例是由容器網路界面 (CNI) 和 kube-proxy 協助程式集啟動的範例。為此，這些 Pod 的規格包含非常寬鬆的容錯能力，以容忍不同的污點。請小心不要變更這些容錯。變更這些容錯可能會導致不正確的叢集操作。此

外，OPA/Gatekeeper 和 Kyverno 等政策管理工具可用於撰寫驗證政策，以防止未經授權的 Pod 使用這些寬鬆容錯。

第 3 部分 - 節點選擇的政策型管理

有數種工具可用來協助管理 Pod 規格的節點親和性與公差，包括強制執行 CICD 管道中的規則。不過，也應在 Kubernetes 叢集層級執行隔離。為此，政策管理工具可用於根據請求承載來變更傳入 Kubernetes API 伺服器請求，以套用上述的個別節點親和性規則和容錯。

例如，以 tenants-x 命名空間為目標的 Pod 可以使用正確的節點親和性和容錯性來加上戳記，以允許在 tenants-x 節點上排程。利用使用 Kubernetes Mutating Admission Webhook 設定的政策管理工具，政策可用來變更傳入 Pod 規格。變動會新增所需的元素，以允許所需的排程。新增節點親和性的範例 OPA/Gatekeeper 政策如下所示。

```
apiVersion: mutations.gatekeeper.sh/v1alpha1
kind: Assign
metadata:
  name: mutator-add-nodeaffinity-pod
  annotations:
    aws-eks-best-practices/description: >-
      Adds Node affinity - https://kubernetes.io/docs/concepts/scheduling-eviction/
assign-pod-node/#node-affinity
spec:
  applyTo:
  - groups: [""]
    kinds: ["Pod"]
    versions: ["v1"]
  match:
    namespaces: ["tenants-x"]
  location:
 "spec.affinity.nodeAffinity.requiredDuringSchedulingIgnoredDuringExecution.nodeSelectorTerms"
  parameters:
    assign:
      value:
        - matchExpressions:
          - key: "tenant"
            operator: In
            values:
            - "tenants-x"
```

上述政策會套用至 Kubernetes API 伺服器請求，以將 Pod 套用至 tenants-x 命名空間。政策會新增`requiredDuringSchedulingIgnoredDuringExecution`節點親和性規則，讓 Pod 被具有`tenant: tenants-x`標籤的節點吸引。

第二個政策如下所示，使用目標命名空間和群組、種類和版本的相同相符條件，將容錯新增至相同的 Pod 規格。

```
apiVersion: mutations.gatekeeper.sh/v1alpha1
kind: Assign
metadata:
  name: mutator-add-toleration-pod
  annotations:
    aws-eks-best-practices/description: >-
      Adds toleration - https://kubernetes.io/docs/concepts/scheduling-eviction/taint-
and-toleration/
spec:
  applyTo:
  - groups: [""]
    kinds: ["Pod"]
    versions: ["v1"]
  match:
    namespaces: ["tenants-x"]
  location: "spec.tolerations"
  parameters:
    assign:
      value:
      - key: "tenant"
        operator: "Equal"
        value: "tenants-x"
        effect: "NoSchedule"
```

上述政策專屬於 Pod；這是因為政策元素中變動`location`元素的路徑所致。您可以撰寫其他政策來處理建立 Pod 的資源，例如部署和任務資源。本指南的配套 GitHub 專案中可以看到列出的政策和其他範例。

這兩個變動的結果是 Pod 會吸引到所需的節點，同時不會被特定節點污點排斥。為了驗證這一點，我們可以看到來自兩個`kubectl`呼叫的輸出片段，以取得標記為 的節點`tenant=tenants-x`，並在`tenants-x`命名空間中取得 Pod。

```
kubectl get nodes -l tenant=tenants-x
NAME
ip-10-0-11-255...
```

```
ip-10-0-28-81...
ip-10-0-43-107...

kubectl -n tenants-x get pods -owide
NAME                                  READY   STATUS    RESTARTS   AGE   IP
 NODE
tenant-test-deploy-58b895ff87-2q7xw   1/1     Running   0          13s   10.0.42.143
 ip-10-0-43-107...
tenant-test-deploy-58b895ff87-9b6hg   1/1     Running   0          13s   10.0.18.145
 ip-10-0-28-81...
tenant-test-deploy-58b895ff87-nxvw5   1/1     Running   0          13s   10.0.30.117
 ip-10-0-28-81...
tenant-test-deploy-58b895ff87-vw796   1/1     Running   0          13s   10.0.3.113
 ip-10-0-11-255...
tenant-test-pod                       1/1     Running   0          13s   10.0.35.83
 ip-10-0-43-107...
```

如以上輸出所示，所有 Pod 都會排程在標有 的節點上`tenant=tenants-x`。簡言之，Pod 只會在所需的節點上執行，而其他 Pod （沒有必要的親和性與容錯能力） 則不會執行。租用戶工作負載已有效隔離。

變動的 Pod 規格範例如下所示。

```
apiVersion: v1
kind: Pod
metadata:
  name: tenant-test-pod
  namespace: tenants-x
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: tenant
            operator: In
            values:
            - tenants-x
...
  tolerations:
  - effect: NoSchedule
    key: tenant
    operator: Equal
```

```
    value: tenants-x
...
```

> **Important**
>
> 與 Kubernetes API 伺服器請求流程整合的政策管理工具，使用變動和驗證許可 Webhook，旨在在特定時間範圍內回應 API 伺服器的請求。這通常為 3 秒或更短。如果 Webhook 呼叫無法在設定的時間內傳回回應，則傳入 API 伺服器請求的變動和/或驗證可能會發生，也可能不會發生。此行為取決於許可 Webhook 組態是設定為 Fail Open 還是 Fail Close。

在上述範例中，我們使用為 OPA/Gatekeeper 撰寫的政策。不過，還有其他政策管理工具也會處理我們的節點選擇使用案例。例如，此 Kyverno 政策可用於處理節點親和性變動。

> **Note**
>
> 如果操作正確，變動政策將影響對傳入 API 伺服器請求承載的所需變更。不過，在允許變更持續之前，也應包含驗證政策，以驗證所需的變更是否發生。這在將這些政策用於tenant-to-node隔離時特別重要。也建議您包含稽核政策，以定期檢查叢集是否有不需要的組態。

參考

- k-rail 旨在協助您透過強制執行特定政策來保護多租戶環境。
- 使用 Amazon EKS 的MultiTenant SaaS 應用程式的安全實務

硬式多租戶

透過為每個租用戶佈建個別叢集，即可實作硬式多租用戶。雖然這在租用戶之間提供了非常強的隔離，但它有幾個缺點。

首先，當您有許多租戶時，這種方法可能會很快變得昂貴。您不僅必須支付每個叢集的控制平面成本，也無法在叢集之間共用運算資源。這最終會導致分段，其中您的叢集子集未充分利用，而其他叢集則過度利用。

其次，您可能需要購買或建置特殊工具來管理所有這些叢集。隨著時間的推移，管理數百個或數千個叢集可能變得過於難以理解。

最後，相對於建立命名空間，為每個租用戶建立叢集的速度會變慢。不過，在高度管制的產業或需要高度隔離的 SaaS 環境中，可能需要硬租用方法。

未來方向

Kubernetes 社群已識別軟式多租用戶目前的缺點，以及硬式多租用戶的挑戰。多租戶特殊興趣群組 (SIG) 正嘗試透過數個孵化專案解決這些缺點，包括階層式命名空間控制器 (HNC) 和虛擬叢集。

HNC 提案 (KEP) 描述了一種在具有 【政策】 物件繼承的命名空間之間建立父子關係的方法，以及租用戶管理員建立子命名空間的能力。

虛擬叢集提案說明為叢集內的每個租用戶 （ 也稱為「Kubernetes on Kubernetes」）建立控制平面服務個別執行個體的機制，包括 API 伺服器、控制器管理員和排程器。

多租戶基準提案提供使用命名空間進行隔離和分割的共享叢集準則，以及驗證是否符合準則的命令列工具 kubectl-mtb。

多叢集管理工具和資源

- Banzai 雲端
- Kommander
- 鏡頭
- 尼馬塔
- 拉法文
- Rancher
- Weave Flux

稽核和記錄

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

收集和分析 【稽核】 日誌對於各種不同的原因很有用。日誌有助於根本原因分析和歸因，即對特定使用者 描述變更。當收集到足夠的日誌時，它們也可用於偵測異常行為。在 EKS 上，稽核日誌會傳送至 Amazon Cloudwatch Logs。EKS 的稽核政策如下所示：

```
apiVersion: audit.k8s.io/v1beta1
kind: Policy
rules:
  # Log full request and response for changes to aws-auth ConfigMap in kube-system
 namespace
  - level: RequestResponse
    namespaces: ["kube-system"]
    verbs: ["update", "patch", "delete"]
    resources:
      - group: "" # core
        resources: ["configmaps"]
        resourceNames: ["aws-auth"]
    omitStages:
      - "RequestReceived"
  # Do not log watch operations performed by kube-proxy on endpoints and services
  - level: None
    users: ["system:kube-proxy"]
    verbs: ["watch"]
    resources:
      - group: "" # core
        resources: ["endpoints", "services", "services/status"]
  # Do not log get operations performed by kubelet on nodes and their statuses
  - level: None
    users: ["kubelet"] # legacy kubelet identity
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["nodes", "nodes/status"]
  # Do not log get operations performed by the system:nodes group on nodes and their
 statuses
  - level: None
    userGroups: ["system:nodes"]
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["nodes", "nodes/status"]
  # Do not log get and update operations performed by controller manager, scheduler,
 and endpoint-controller on endpoints in kube-system namespace
  - level: None
    users:
      - system:kube-controller-manager
      - system:kube-scheduler
      - system:serviceaccount:kube-system:endpoint-controller
```

```
    verbs: ["get", "update"]
    namespaces: ["kube-system"]
    resources:
      - group: "" # core
        resources: ["endpoints"]
  # Do not log get operations performed by apiserver on namespaces and their statuses/
finalizations
  - level: None
    users: ["system:apiserver"]
    verbs: ["get"]
    resources:
      - group: "" # core
        resources: ["namespaces", "namespaces/status", "namespaces/finalize"]
  # Do not log get and list operations performed by controller manager on
 metrics.k8s.io resources
  - level: None
    users:
      - system:kube-controller-manager
    verbs: ["get", "list"]
    resources:
      - group: "metrics.k8s.io"
  # Do not log access to health, version, and swagger non-resource URLs
  - level: None
    nonResourceURLs:
      - /healthz*
      - /version
      - /swagger*
  # Do not log events resources
  - level: None
    resources:
      - group: "" # core
        resources: ["events"]
  # Log request for updates/patches to nodes and pods statuses by kubelet and node
 problem detector
  - level: Request
    users: ["kubelet", "system:node-problem-detector", "system:serviceaccount:kube-
system:node-problem-detector"]
    verbs: ["update", "patch"]
    resources:
      - group: "" # core
        resources: ["nodes/status", "pods/status"]
    omitStages:
      - "RequestReceived"
  # Log request for updates/patches to nodes and pods statuses by system:nodes group
```

```
  - level: Request
    userGroups: ["system:nodes"]
    verbs: ["update", "patch"]
    resources:
      - group: "" # core
        resources: ["nodes/status", "pods/status"]
    omitStages:
      - "RequestReceived"
  # Log delete collection requests by namespace-controller in kube-system namespace
  - level: Request
    users: ["system:serviceaccount:kube-system:namespace-controller"]
    verbs: ["deletecollection"]
    omitStages:
      - "RequestReceived"
  # Log metadata for secrets, configmaps, and tokenreviews to protect sensitive data
  - level: Metadata
    resources:
      - group: "" # core
        resources: ["secrets", "configmaps"]
      - group: authentication.k8s.io
        resources: ["tokenreviews"]
    omitStages:
      - "RequestReceived"
  # Log requests for serviceaccounts/token resources
  - level: Request
    resources:
      - group: "" # core
        resources: ["serviceaccounts/token"]
  # Log get, list, and watch requests for various resource groups
  - level: Request
    verbs: ["get", "list", "watch"]
    resources:
      - group: "" # core
      - group: "admissionregistration.k8s.io"
      - group: "apiextensions.k8s.io"
      - group: "apiregistration.k8s.io"
      - group: "apps"
      - group: "authentication.k8s.io"
      - group: "authorization.k8s.io"
      - group: "autoscaling"
      - group: "batch"
      - group: "certificates.k8s.io"
      - group: "extensions"
      - group: "metrics.k8s.io"
```

```
      - group: "networking.k8s.io"
      - group: "policy"
      - group: "rbac.authorization.k8s.io"
      - group: "scheduling.k8s.io"
      - group: "settings.k8s.io"
      - group: "storage.k8s.io"
    omitStages:
      - "RequestReceived"
  # Default logging level for known APIs to log request and response
  - level: RequestResponse
    resources:
      - group: "" # core
      - group: "admissionregistration.k8s.io"
      - group: "apiextensions.k8s.io"
      - group: "apiregistration.k8s.io"
      - group: "apps"
      - group: "authentication.k8s.io"
      - group: "authorization.k8s.io"
      - group: "autoscaling"
      - group: "batch"
      - group: "certificates.k8s.io"
      - group: "extensions"
      - group: "metrics.k8s.io"
      - group: "networking.k8s.io"
      - group: "policy"
      - group: "rbac.authorization.k8s.io"
      - group: "scheduling.k8s.io"
      - group: "settings.k8s.io"
      - group: "storage.k8s.io"
    omitStages:
      - "RequestReceived"
  # Default logging level for all other requests to log metadata only
  - level: Metadata
    omitStages:
      - "RequestReceived"
```

建議

啟用稽核日誌

稽核日誌是 EKS 管理的 EKS 受管 Kubernetes 控制平面日誌的一部分。如需啟用/停用控制平面日誌的指示，包括 Kubernetes API 伺服器、控制器管理員和排程器的日誌，以及稽核日誌，請參閱

https：//https://docs.aws.amazon.com/eks/latest/userguide/control-plane-logs.html#enabling-control-plane-log-export。

> **Note**
>
> 當您啟用控制平面記錄時，在 CloudWatch 中存放日誌會產生費用。這引發了有關持續安全成本的更廣泛的問題。最後，您必須權衡這些成本與安全違規的成本，例如 財務損失、聲譽受損等。您可能會發現，您只能實作本指南中的一些建議，以充分保護您的環境。

> **Warning**
>
> CloudWatch Logs 項目的大小上限為 1MB，Kubernetes API 請求大小上限為 1.5MiB。大於 1MB 的日誌項目將被截斷，或僅包含請求中繼資料。

使用稽核中繼資料

Kubernetes 稽核日誌包含兩個註釋，指出請求是否獲得授權，`authorization.k8s.io/decision`以及決策的原因`authorization.k8s.io/reason`。使用這些屬性來確定允許特定 API 呼叫的原因。

建立可疑事件的警示

建立警示，在 403 禁止和 401 未經授權的回應增加時自動提醒您，然後使用 `host`、 `sourceIPs`和等屬性`k8s_user.username`來找出這些請求的來源。

使用 Log Insights 分析日誌

使用 CloudWatch Log Insights 監控 RBAC 物件的變更，例如 角色、RoleBindings、ClusterRoles 和 ClusterRoleBindings。以下顯示一些範例查詢：

列出 ConfigMap `aws-auth` 的更新：

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter verb in ["update", "patch"]
| filter objectRef.resource = "configmaps" and objectRef.name = "aws-auth" and
 objectRef.namespace = "kube-system"
```

```
| sort @timestamp desc
```

列出驗證 Webhook 的新 或變更的建立：

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter verb in ["create", "update", "patch"] and responseStatus.code = 201
| filter objectRef.resource = "validatingwebhookconfigurations"
| sort @timestamp desc
```

列出角色的建立、更新、刪除操作：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="roles" and verb in ["create", "update", "patch", "delete"]
```

列出 RoleBindings 的建立、更新、刪除操作：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="rolebindings" and verb in ["create", "update", "patch",
 "delete"]
```

列出 ClusterRoles 的建立、更新、刪除操作：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="clusterroles" and verb in ["create", "update", "patch",
 "delete"]
```

列出 ClusterRoleBindings 的建立、更新、刪除操作：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="clusterrolebindings" and verb in ["create", "update",
 "patch", "delete"]
```

針對秘密繪製未經授權的讀取操作：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter objectRef.resource="secrets" and verb in ["get", "watch", "list"] and
 responseStatus.code="401"
| stats count() by bin(1m)
```

失敗的匿名請求清單：

```
fields @timestamp, @message, sourceIPs.0
| sort @timestamp desc
| limit 100
| filter user.username="system:anonymous" and responseStatus.code in ["401", "403"]
```

稽核 CloudTrail 日誌

使用服務帳戶 IAM 角色 (IRSA) 的 Pod 呼叫的 AWS APIs 會自動記錄到 CloudTrail 和服務帳戶的名稱。如果未明確授權呼叫 API 的服務帳戶名稱出現在日誌中，可能表示 IAM 角色的信任政策設定錯誤。一般而言，Cloudtrail 是對特定 IAM 主體進行 AWS API 呼叫的絕佳方式。

使用 CloudTrail Insights 來偵錯可疑活動

CloudTrail 洞察會自動分析來自 CloudTrail 追蹤的寫入管理事件，並提醒您異常活動。這可協助您識別 AWS 帳戶中寫入 APIs的呼叫量何時增加，包括來自使用 IRSA 擔任 IAM 角色的 Pod。如需詳細資訊，請參閱宣布 CloudTrail Insights：識別和回應異常 API 活動。

其他資源

隨著日誌數量的增加，使用 Log Insights 或其他日誌分析工具剖析和篩選日誌可能會變得無效。或者，建議您考慮執行 Sysdig Falco 和 ekscloudwatch。Falco 會分析稽核日誌，並標記長時間的異常或濫用。ekscloudwatch 專案會將稽核日誌事件從 CloudWatch 轉送至 Falco 進行分析。Falco 提供一組預設稽核規則，以及新增您自己的功能。

不過，另一個選項可能是將稽核日誌存放在 S3 中，並使用 SageMaker 隨機剪切森林演算法來異常行為，需要進一步調查。

工具和資源

下列商業和開放原始碼專案可用來評估叢集與既定最佳實務的一致性：

- Amazon EKS 安全浸入研討會 - Detective 控制
- kubeaudit
- kube-scan 根據 Kubernetes 常用組態評分系統架構，為叢集中執行的工作負載指派風險分數
- kubesec.io
- 極性
- 右舷
- Snyk
- Kubescape Kubescape 是一種開放原始碼 kubernetes 安全工具，可掃描叢集、YAML 檔案和 Helm Chart。它根據多個架構（包括 NSA-CISA 和 MITRE ATT&CK®) 偵測錯誤組態。

網路安全

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

網路安全有幾個面向。第一個涉及規則的套用，這些規則會限制服務之間的網路流量流程。第二個涉及傳輸中流量的加密。在 EKS 上實作這些安全措施的機制會有所不同，但通常包括下列項目：

流量控制

- 網路政策
- 安全群組

網路加密

- Service Mesh
- 輸入控制器和負載平衡器
- Nitro 執行個體
- 使用 cert-manager 的 ACM Private CA

網路政策

在 Kubernetes 叢集中，預設允許所有 Pod 對 Pod 通訊。雖然這種靈活性可能有助於促進實驗，但它不被視為安全。Kubernetes 網路政策為您提供一種機制，以限制 Pod 之間的網路流量（通常稱為東部/西部流量），以及 Pod 與外部服務之間的網路流量。Kubernetes 網路政策在 OSI 模型的第 3 層和第 4 層運作。網路政策使用 Pod、命名空間選擇器和標籤來識別來源和目的地 Pod，但也可以包含 IP 地址、連接埠號碼、通訊協定或這些項目的組合。網路政策可套用至 Pod 的傳入或傳出連線，通常稱為傳入和傳出規則。

透過 Amazon VPC CNI 外掛程式的原生網路政策支援，您可以實作網路政策來保護 kubernetes 叢集中的網路流量。這與上游 Kubernetes 網路政策 API 整合，以確保相容性和遵守 Kubernetes 標準。您可以使用上游 API 支援的不同識別符來定義政策。根據預設，允許所有傳入和傳出流量到 Pod。指定具有 policyType Ingress 的網路政策時，只有來自 Pod 節點的允許連線，以及輸入規則允許的連線。輸出規則也適用。如果定義了多個規則，則在做出決策時會將所有規則的聯集納入考量。因此，評估順序不會影響政策結果。

> **Important**
>
> 當您第一次佈建 EKS 叢集時，預設不會啟用 VPC CNI 網路政策功能。請確定您已部署支援的 VPC CNI 附加元件版本，並在 vpc-cni 附加元件`true`上將`ENABLE_NETWORK_POLICY`旗標設定為 以啟用此功能。如需詳細說明，請參閱 Amazon EKS 使用者指南。

建議

網路政策入門 - 遵循最低權限原則

建立預設拒絕政策

如同 RBAC 政策，建議使用網路政策遵循最低權限存取原則。首先建立拒絕所有限制命名空間中所有傳入和傳出流量的政策。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
  namespace: default
spec:
  podSelector: {}
  policyTypes:
```

```
  - Ingress
  - Egress
```

預設拒絕

		Target		Source				
#	Policy name	Namespace	Pods	Namespace	Pods	Subnet	Ports	
1	default-deny	default	Any	Any	Any	Any	Any	

> **Note**
>
> 上圖是由網路政策檢視器從 Tufin 建立。

建立規則以允許 DNS 查詢

設定預設拒絕所有規則後，您就可以開始分層其他規則，例如允許 Pod 查詢 CoreDNS 以進行名稱解析的規則。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns-access
  namespace: default
spec:
  podSelector:
    matchLabels: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          kubernetes.io/metadata.name: kube-system
      podSelector:
        matchLabels:
          k8s-app: kube-dns
    ports:
    - protocol: UDP
      port: 53
```

allow-dns-access

		Target		Destination				
#	Policy name	Namespace	Pods	Namespace	Pods	Subnet	Ports	
1	allow-dns-access	default	Any	name: kube-system	Any		UDP: 53	

遞增新增規則，以選擇性地允許命名空間/Pod 之間的流量

了解應用程式需求，並視需要建立精細的輸入和輸出規則。以下範例示範如何將連接埠 80 上的輸入流量app-one從 限制為 client-one。這有助於將攻擊面降至最低，並降低未經授權的存取風險。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress-app-one
  namespace: default
spec:
  podSelector:
    matchLabels:
      k8s-app: app-one
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          k8s-app: client-one
    ports:
    - protocol: TCP
      port: 80
```

allow-ingress-app-one

		Target		Source				
#	Policy name	Namespace	Pods	Namespace	Pods	Subnet	Ports	
1	allow-ingress-app-one	default	k8s-app: app-one	default	k8s-app: client-one		TCP: 80	

監控網路政策強制執行

- 使用網路政策編輯器
  - 網路政策編輯器有助於視覺化、安全分數、從網路流量日誌自動產生

  - 以互動方式建置網路政策
- 稽核日誌
  - 定期檢閱 EKS 叢集的稽核日誌
  - 稽核日誌提供有關叢集上執行哪些動作的豐富資訊，包括網路政策的變更
  - 使用此資訊來追蹤網路政策隨著時間的變更，並偵測任何未經授權或非預期的變更
- 自動化測試
  - 建立模擬生產環境的測試環境，並定期部署嘗試違反網路政策的工作負載，以實作自動化測試。
- 監控指標
  - 設定可觀測性代理程式，從 VPC CNI 節點代理程式抓取 prometheus 指標，允許 監控代理程式運作狀態和 sdk 錯誤。
- 定期稽核網路政策
  - 定期稽核您的網路政策，以確保它們符合您目前的應用程式需求。隨著應用程式的演進，稽核讓您有機會移除備援輸入、輸出規則，並確保您的應用程式沒有過多的許可。
- 使用開放政策代理程式 (OPA) 確保網路政策存在
  - 使用如下所示的 OPA 政策，以確保網路政策在加入應用程式 Pod 之前始終存在。`k8s-app: sample-app` 如果對應的網路政策不存在，此政策會拒絕使用標籤加入 k8s Pod。

```
package kubernetes.admission
import data.kubernetes.networkpolicies

deny[msg] {
    input.request.kind.kind == "Pod"
    pod_label_value := {v["k8s-app"] | v := input.request.object.metadata.labels}
    contains_label(pod_label_value, "sample-app")
    np_label_value := {v["k8s-app"] | v :=
 networkpolicies[_].spec.podSelector.matchLabels}
    not contains_label(np_label_value, "sample-app")
    msg:= sprintf("The Pod %v could not be created because it is missing an associated
 Network Policy.", [input.request.object.metadata.name])
}
contains_label(arr, val) {
    arr[_] == val
}
```

疑難排解

監控 vpc-network-policy-controller、 node-agent 日誌

啟用 EKS 控制平面控制器管理員日誌來診斷網路政策功能。您可以將控制平面日誌串流至 CloudWatch 日誌群組，並使用 CloudWatch Log Insights 來執行進階查詢。從日誌中，您可以檢視哪些 Pod 端點物件解析為網路政策、政策的調校狀態，以及政策是否如預期般運作時的偵錯。

此外，Amazon VPC CNI 可讓您從 EKS 工作者節點啟用政策強制執行日誌的收集和匯出至 Amazon Cloudwatch。啟用後，您可以利用 CloudWatch Container Insights 提供與網路政策相關的用量洞察。

Amazon VPC CNI 也提供 SDK，提供與節點上 eBPF 程式互動的界面。將 部署到節點時`aws-node`，會安裝 SDK。您可以在節點的 `/opt/cni/bin`目錄下找到安裝的 SDK 二進位檔。在啟動時，軟體開發套件支援基本功能，例如檢查 eBPF 程式和映射。

```
sudo /opt/cni/bin/aws-eks-na-cli ebpf progs
```

日誌網路流量中繼資料

AWS VPC Flow Logs 會擷取流經 VPC 之流量的中繼資料，例如來源和目的地 IP 地址、連接埠，以及接受/捨棄的封包。此資訊可以進行分析，以尋找 VPC 內資源之間的可疑或異常活動，包括 Pod。不過，由於 Pod 的 IP 地址在取代時經常變更，因此流程日誌本身可能不夠。Calico Enterprise 使用 Pod 標籤和其他中繼資料擴展流程日誌，讓您更輕鬆地解密 Pod 之間的流量流程。

Security groups (安全群組)

EKS 使用 AWS VPC 安全群組 (SGs) 來控制 Kubernetes 控制平面與叢集工作者節點之間的流量。安全群組也用於控制工作者節點與其他 VPC 資源之間的流量，以及外部 IP 地址。當您佈建 EKS 叢集（使用 Kubernetes 1.14-eks.3 版或更新版本）時，系統會自動為您建立叢集安全群組。此安全群組允許 EKS 控制平面與受管節點群組節點之間的不中斷通訊。為了簡化，建議您將叢集 SG 新增至所有節點群組，包括未受管節點群組。

在 Kubernetes 版本 1.14 和 EKS 版本 eks.3 之前，針對 EKS 控制平面和節點群組設定了個別的安全群組。您可以在 https：//https://docs.aws.amazon.com/eks/latest/userguide/sec-group-reqs.html 找到控制平面和節點群組安全群組的最低和建議規則。控制平面安全群組的最低規則允許連接埠 443 從工作者節點 SG 傳入。此規則允許 kubelet 與 Kubernetes API 伺服器通訊。它還包括傳出流量到工作者節點 SG 的連接埠 10250；10250 是 kubelet 接聽的連接埠。同樣地，最小節點群組規則允許連接埠 10250 從控制平面 SG 傳入，以及 443 傳出至控制平面 SG。最後有一個規則允許節點群組內的節點之間進行不中斷的通訊。

如果您需要控制叢集內執行的服務與叢集外執行的服務之間的通訊，例如 RDS 資料庫，請考慮 Pod 的安全群組。透過 Pod 的安全群組，您可以將現有的安全群組指派給 Pod 集合。

> **Warning**
>
> 如果您參考在建立 Pod 之前不存在的安全群組，則不會排程 Pod。

您可以透過建立`SecurityGroupPolicy`物件並指定 `PodSelector`或 來控制指派給安全群組的 `PodServiceAccountSelector`。將選取器設定為 `{}`會將 中參考SGs 指派給命名空間中的所有 `SecurityGroupPolicy` Pod 或命名空間中的所有服務帳戶。在實作 Pod 的安全群組之前，請確定您已熟悉所有考量事項。

> **Important**
>
> 如果您將 SGs用於 Pod，則必須建立允許連接埠 53 傳出至叢集安全群組的 SGs。同樣地，您必須更新叢集安全群組，以接受來自 Pod 安全群組的連接埠 53 傳入流量。

> **Important**
>
> 對 Pod 使用安全群組時，安全群組的限制仍然適用，因此請謹慎使用。

> **Important**
>
> 您必須為針對 Pod 設定的所有探查，從叢集安全群組 (kubelet) 建立傳入流量的規則。

> **Important**
>
> Pod 的安全群組依賴於稱為 ENI 中繼的功能，該功能已建立以增加 EC2 執行個體的 ENI 密度。當 Pod 指派給 SG 時，VPC 控制器會將來自節點群組的分支 ENI 與 Pod 建立關聯。如果在排程 Pod 時節點群組中沒有足夠的分支 ENIs，則 Pod 將保持待定狀態。執行個體可支援的分支 ENIs 數量會因執行個體類型/系列而異。如需更多詳細資訊，請參閱 https://docs.aws.amazon.com/eks/latest/userguide/security-groups-for-pods.html#supported-instance-types：//。

雖然 Pod 的安全群組提供 AWS 原生的方式來控制叢集內外的網路流量，而不會產生政策協助程式的額外負荷，但還有其他選項可用。例如，Cilium 政策引擎可讓您在網路政策中參考 DNS 名稱。Calico Enterprise 包含將網路政策映射至 AWS 安全群組的選項。如果您已實作類似 Istio 的服務網格，您可以使用輸出閘道，將網路輸出限制為特定、完全合格的網域或 IP 地址。如需此選項的詳細資訊，請閱讀 Istio 中輸出流量控制的三個部分系列。

何時使用網路政策與 Pod 的安全群組？

何時使用 Kubernetes 網路政策

- 控制 pod-to-pod流量
  - 適用於控制叢集內 Pod 之間的網路流量 （ 東西流量 ）
- 在 IP 地址或連接埠層級控制流量 (OSI layer 3 或 4)

何時針對 Pod 使用 AWS 安全群組 (SGP)

- 利用現有的 AWS 組態
  - 如果您已經有一組複雜的 EC2 安全群組來管理對 AWS 服務的存取，而且您正在將應用程式從 EC2 執行個體遷移到 EKS，則 SGPs 可以是一個很好的選擇，允許您重複使用安全群組資源並將其套用到您的 Pod。
- 控制對 AWS 服務的存取
  - 您在 EKS 叢集中執行的應用程式想要與其他 AWS 服務 (RDS 資料庫 ） 通訊，請使用 SGPs做為有效的機制，以控制從 Pod 到 AWS 服務的流量。
- Pod 和節點流量的隔離
  - 如果您想要將 Pod 流量與其他節點流量完全分開，請在 `POD_SECURITY_GROUP_ENFORCING_MODE=strict` 模式中使用 SGP。

使用 Pod 安全群組和網路政策的最佳實務

- 分層安全性
  - 針對分層安全方法使用 SGP 和 kubernetes 網路政策的組合
  - 使用 SGPs限制網路層級存取不屬於叢集的 AWS 服務，而 kubernetes 網路政策可以限制叢集內 Pod 之間的網路流量
- 最低權限原則
  - 僅允許 Pod 或命名空間之間的必要流量

- 分割您的應用程式
  - 盡可能依網路政策分割應用程式，以便在應用程式遭到入侵時減少爆量半徑
- 保持政策簡單明瞭
  - Kubernetes 網路政策可以相當精細且複雜，最好盡可能保持簡單，以降低組態錯誤的風險並減輕管理開銷
- 減少攻擊面
  - 透過限制應用程式的暴露，將攻擊面降至最低

> **Important**
>
> Pod 的安全群組提供兩種強制執行模式： `strict`和 `standard`。在 EKS 叢集中使用網路政策和 Pod 功能的安全群組時，您必須使用 `standard` 模式。

在網路安全方面，分層方法通常是最有效的解決方案。搭配使用 kubernetes 網路政策和 SGP 可為在 EKS 中執行的應用程式提供強大的defense-in-depth策略。

Service Mesh 政策強制執行或 Kubernetes 網路政策

`service mesh` 是您可以新增至應用程式的專用基礎設施層。它可讓您透明地新增可觀測性、流量管理和安全性等功能，而無需將其新增至您自己的程式碼。

服務網格會在 OSI 模型的第 7 層（應用程式）強制執行政策，而 kubernetes 網路政策則會在第 3 層（網路）和第 4 層（傳輸）操作。此空間中有許多方案，例如 AWS AppMesh、Istio、Linkerd 等。

何時使用服務網格執行政策

- 對服務網格進行現有投資
- 需要更進階的功能，例如流量管理、可觀測性和安全性
  - 流量控制、負載平衡、電路中斷、速率限制、逾時等。
  - 服務執行方式的詳細洞見（延遲、錯誤率、每秒請求數、請求量等）
  - 您想要針對 mTLS 等安全功能實作和利用服務網格

針對更簡單的使用案例選擇 Kubernetes 網路政策

- 限制哪些 Pod 可以互相通訊

- 網路政策所需的資源少於服務網格，因此非常適合較簡單的使用案例，或是執行和管理服務網格的額外負荷可能不合理的較小叢集

> **Note**
>
> 網路政策和服務網格也可以一起使用。使用網路政策在 Pod 之間提供基準層級的安全性和隔離，然後使用服務網格來新增其他功能，例如流量管理、可觀測性和安全性。

ThirdParty網路政策引擎

當您有進階政策需求，例如全球網路政策、支援以 DNS 主機名稱為基礎的規則、第 7 層規則、以 ServiceAccount 為基礎的規則，以及明確拒絕/記錄動作等時，請考慮第三方網路政策引擎。Calico 是來自 Tigera 的開放原始碼政策引擎，可與 EKS 搭配使用。除了實作完整的 Kubernetes 網路政策功能，Calico 還支援具有更豐富功能的網路政策，包括支援第 7 層規則，例如 HTTP，當 與 Istio 整合時。Calico 政策的範圍可以是命名空間、Pod、服務帳戶或全域。當政策範圍限定於服務帳戶時，它會將一組輸入/輸出規則與該服務帳戶建立關聯。透過適當的 RBAC 規則，您可以防止團隊覆寫這些規則，讓 IT 安全專業人員安全地委派命名空間的管理。Cilium 的維護者 Isovalent 也已擴展網路政策，以包含對第 7 層規則的部分支援，例如 HTTP。Cilium 也支援 DNS 主機名稱，有助於限制 Kubernetes Services/Pod 與 VPC 內外執行的資源之間的流量。相反地，Calico Enterprise 包含的功能可讓您將 Kubernetes 網路政策映射至 AWS 安全群組，以及 DNS 主機名稱。

您可以在 https://github.com/ahmetb/kubernetes-network-policy-recipes：// 找到常見的 Kubernetes 網路政策清單。如需 Calico 的類似規則集，請參閱 https：//https://docs.projectcalico.org/security/calico-network-policy。

遷移至 Amazon VPC CNI 網路政策引擎

為了保持一致性並避免意外的 Pod 通訊行為，建議您在叢集中僅部署一個網路政策引擎。如果您想要從 3P 遷移至 VPC CNI 網路政策引擎，建議您先將現有的 3P NetworkPolicy CRDs轉換為 Kubernetes NetworkPolicy 資源，再啟用 VPC CNI 網路政策支援。此外，請在單獨的測試叢集中測試遷移的政策，然後再將其套用至您的生產環境。這可讓您識別和解決 Pod 通訊行為中的任何潛在問題或不一致。

遷移工具

為了協助您進行遷移程序，我們開發了一個名為 K8s Network Policy Migrator 的工具，可將您現有的 Calico/Cilium 網路政策 CRDs 轉換為 Kubernetes 原生網路政策。轉換後，您可以在執行 VPC CNI 網

路政策控制器的新叢集上直接測試轉換的網路政策。此工具旨在協助您簡化遷移程序，並確保順利轉換。

> **Important**
>
> 遷移工具只會轉換與原生 kubernetes 網路政策 api 相容的 3P 政策。如果您使用 3P 外掛程式提供的進階網路政策功能，遷移工具會略過並報告它們。

請注意，AWS VPC CNI 網路政策工程團隊目前不支援遷移工具，該工具會盡力提供給客戶。我們鼓勵您利用此工具來促進遷移程序。如果您遇到工具的任何問題或錯誤，我們歡迎您建立 GitHub 問題。您的意見回饋對我們來說非常寶貴，有助於持續改善我們的服務。

其他資源

- Kubernetes & Tigera：網路政策、安全性和稽核
- Calico Enterprise
- Cilium
- NetworkPolicy 編輯器 Cilium 的互動式政策編輯器
- Inspektor Gadget 建議網路政策小工具 根據網路流量分析建議網路政策

傳輸中加密

需要符合 PCI、HIPAA 或其他法規的應用程式，可能需要在傳輸時加密資料。現在 TLS 是加密線路上流量的事實選擇。TLS 如同前一代 SSL，使用密碼編譯通訊協定透過網路提供安全通訊。TLS 使用對稱加密，其中加密資料的金鑰是根據工作階段開始時交涉的共用秘密產生。以下是您可以在 Kubernetes 環境中加密資料的幾種方式。

Nitro 執行個體

根據預設，下列 Nitro 執行個體類型之間交換的流量會自動加密，例
如 C5n、G4、I3en、M5dn、M5n、P3dn、R5dn 和 R5n。有傳輸閘道或負載平衡器等中繼躍點時，不會加密流量。如需傳輸中加密的更多詳細資訊，以及預設支援網路加密的執行個體類型完整清單，請參閱傳輸中加密。

Service Mesh

傳輸中加密也可以使用服務網格實作，例如 App Mesh、Linkerd v2 和 Istio。AppMesh 支援具有 X.509 憑證或 Envoy 秘密探索服務 (SDS) 的 mTLS。Linkerd 和 Istio 都支援 mTLS。

aws-app-mesh-examples GitHub 儲存庫提供使用 X.509 憑證設定 mTLS 的逐步解說，以及搭配 Envoy 容器做為 SDS 供應商的 SPIRE：

- 使用 X.509 憑證設定 mTLS
- 使用 SPIRE (SDS) 設定 TLS

App Mesh 也支援使用 AWS Certificate Manager (ACM) 發行的私有憑證或存放在虛擬節點本機檔案系統的憑證進行 TLS 加密。

aws-app-mesh-examples GitHub 儲存庫提供逐步解說，以使用 ACM 發行的憑證和 Envoy 容器隨附的憑證來設定 TLS：

- 使用檔案提供的 TLS 憑證設定 TLS
- 使用 AWS Certificate Manager 設定 TLS

輸入控制器和負載平衡器

輸入控制器可讓您以智慧方式將從叢集外部發出的 HTTP/S 流量路由至叢集內執行的服務。通常，這些傳入會由第 4 層負載平衡器前置，例如 Classic Load Balancer 或 Network Load Balancer (NLB)。加密的流量可以在網路中的不同位置終止，例如 負載平衡器、傳入資源或 Pod。終止 SSL 連線的方式和位置最終將由組織的網路安全政策決定。例如，如果您的政策需要end-to-end加密，您必須解密 Pod 的流量。這會對 Pod 造成額外的負擔，因為它必須花費週期來建立初始交握。整體 SSL/TLS 處理非常耗用 CPU。因此，如果您有彈性，請嘗試在輸入或負載平衡器執行 SSL 卸載。

搭配 AWS Elastic Load Balancer 使用加密

AWS Application Load Balancer (ALB) 和 Network Load Balancer (NLB) 都支援傳輸加密 (SSL 和 TLS)。ALB 的`alb.ingress.kubernetes.io/certificate-arn`註釋可讓您指定要新增至 ALB 的憑證。如果您省略註釋，控制器會嘗試透過使用主機欄位比對可用的 AWS Certificate Manager (ACM) 憑證，將憑證新增至需要它的接聽程式。從 EKS 1.15 版開始，您可以搭配 NLB 使用`service.beta.kubernetes.io/aws-load-balancer-ssl-cert`註釋，如以下範例所示。

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: demo-app
  namespace: default
  labels:
    app: demo-app
  annotations:
     service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
     service.beta.kubernetes.io/aws-load-balancer-ssl-cert: "<certificate ARN>"
     service.beta.kubernetes.io/aws-load-balancer-ssl-ports: "443"
     service.beta.kubernetes.io/aws-load-balancer-backend-protocol: "http"
spec:
  type: LoadBalancer
  ports:
  - port: 443
    targetPort: 80
    protocol: TCP
  selector:
    app: demo-app
//---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: nginx
  namespace: default
  labels:
    app: demo-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo-app
  template:
    metadata:
      labels:
        app: demo-app
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 443
              protocol: TCP
            - containerPort: 80
```

```
            protocol: TCP
```

以下是 SSL/TLS 終止的其他範例。

- 使用輪廓保護 EKS 傳入，讓我們加密 GitOps 方法
- 如何使用 ACM 終止 Amazon EKS 工作負載上的 HTTPS 流量？

> **Important**
>
> 有些輸入，例如 AWS LB 控制器，使用註釋實作 SSL/TLS，而不是作為輸入規格的一部分。

使用 cert-manager 的 ACM Private CA

您可以啟用 TLS 和 mTLS，使用 ACM Private Certificate Authority (CA) 和 cert-manager 保護輸入、Pod 以及 Pod 之間的 EKS 應用程式工作負載，這是分發、續約和撤銷憑證的熱門 Kubernetes 附加元件。ACM Private CA 是高可用性、安全、受管的 CA，無需管理您自己的 CA 的預付和維護成本。如果您使用的是預設的 Kubernetes 憑證授權單位，則有機會透過 ACM Private CA 提高安全性並滿足合規要求。ACM Private CA 會保護 FIPS 140-2 第 3 級硬體安全模組中的私有金鑰（非常安全），而預設的 CA 會存放以記憶體編碼的金鑰（較不安全）。集中式 CA 也可讓您在 Kubernetes 環境內外，更進一步控制並改善私有憑證的可稽核性。

工作負載之間相互 TLS 的短期 CA 模式

在 EKS 中使用 ACM Private CA for mTLS 時，建議您使用具有短期 CA 模式的短期有效憑證。雖然可以在一般用途 CA 模式中發行短期憑證，但對於需要頻繁發行新憑證的使用案例，使用短期 CA 模式可提高成本效益（比一般模式便宜約 75%)。此外，您應該嘗試將私有憑證的有效期間與 EKS 叢集中的 Pod 生命週期保持一致。在此處進一步了解 ACM Private CA 及其優點。

ACM 設定說明

從建立私有 CA 開始，請遵循 ACM Private CA 技術文件中提供的程序。一旦您擁有私有 CA，請使用一般安裝指示來安裝 cert-manager。安裝 cert-manager 之後，請依照 GitHub 中的設定說明安裝 Private CA Kubernetes cert-manager 外掛程式。外掛程式可讓 cert-manager 從 ACM Private CA 請求私有憑證。

現在您已擁有私有 CA 和已安裝 cert-manager 和 外掛程式的 EKS 叢集，是時候設定許可並建立發行者了。更新 EKS 節點角色的 IAM 許可，以允許存取 ACM Private CA。將 取代<CA_ARN>為您的私有 CA 的值：

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "awspcaissuer",
            "Action": [
                "acm-pca:DescribeCertificateAuthority",
                "acm-pca:GetCertificate",
                "acm-pca:IssueCertificate"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:acm-pca:us-west-2:123456789012:certificate-
authority/12345678-1234-1234-1234-123456789012"
        }
    ]
}
```

也可以使用 IAM 帳戶或 IRSA 的服務角色。如需完整範例，請參閱下列其他資源一節。

在 Amazon EKS 中建立發行者，方法是使用下列文字建立名為 cluster-issuer.yaml 的自訂資源定義檔案，以私有 CA 取代 <CA_ARN>和 <Region>資訊。

```
apiVersion: awspca.cert-manager.io/v1beta1
kind: AWSPCAClusterIssuer
metadata:
          name: demo-test-root-ca
spec:
          arn: <CA_ARN>
          region: <Region>
```

部署您建立的發行者。

```
kubectl apply -f cluster-issuer.yaml
```

EKS 叢集已設定為向 Private CA 請求憑證。您現在可以使用 cert-manager Certificate 的資源，透過將issuerRef欄位的值變更為您在上面建立的私有 CA 發行者來發行憑證。如需如何指定和請求憑證資源的詳細資訊，請參閱 cert-manager 的憑證資源指南。請參閱此處的範例。

具有 Istio 和 cert-manager 的 ACM Private CA

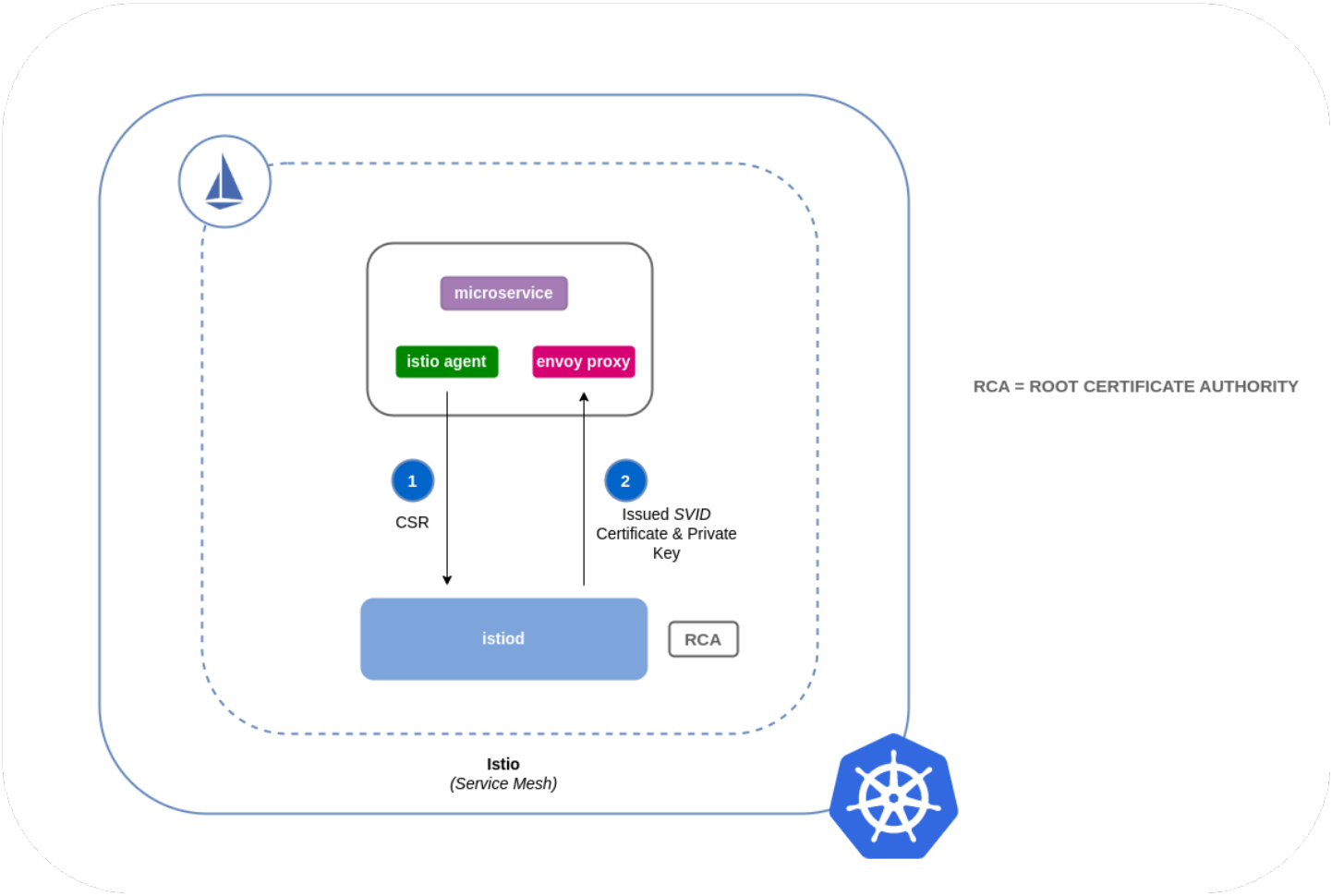
如果您在 EKS 叢集中執行 Istio，您可以停用 Istio 控制平面 （特別是 `istiod`) 做為根憑證授權單位 (CA)，並將 ACM Private CA 設定為工作負載之間 mTLS 的根 CA。如果您要使用此方法，請考慮在 ACM Private CA 中使用短期 CA 模式。如需詳細資訊，請參閱上一節和此部落格文章。

憑證簽署如何在 Istio 中運作 （預設）

Kubernetes 中的工作負載是使用服務帳戶來識別。如果您未指定服務帳戶，Kubernetes 會自動將服務帳戶指派給工作負載。此外，服務帳戶會自動掛載相關聯的字符。服務帳戶會使用此權杖，讓工作負載針對 Kubernetes API 進行身分驗證。服務帳戶可能足以做為 Kubernetes 的身分，但 Istio 擁有自己的身分管理系統和 CA。當工作負載以其呼叫附屬代理啟動時，它需要從 Istio 指派的身分，以便將其視為可信，並允許與網格中的其他 服務通訊。

若要從 Istio 取得此身分， `istio-agent`會將稱為憑證簽署請求 （或 CSR) 的請求傳送至 Istio 控制平面。此 CSR 包含服務帳戶字符，以便在處理之前驗證工作負載的身分。此驗證程序由 處理`istiod`，同時做為註冊授權單位 （或 RA) 和 CA。RA 做為gatekeeper，確保只有已驗證的 CSR 才能將其傳遞給 CA。驗證 CSR 後，該 CSR 會轉送至 CA，然後向服務帳戶發行包含 SPIFFE 身分的憑證。此憑證稱為 SPIFFE 驗證身分文件 （或 SVID)。SVID 會指派給請求的服務，以用於識別和加密通訊服務之間傳輸的流量。

Istio 憑證簽署請求的預設流程：



憑證簽署如何搭配 ACM Private CA 在 Istio 中運作

您可以使用稱為 Istio 憑證簽署請求代理程式 (istio-csr) 的 cert-manager 附加元件，將 Istio 與 ACM Private CA 整合。此代理程式允許使用憑證管理員發行者保護 Istio 工作負載和控制平面元件，在此情況下為 ACM Private CA。istio-csr 代理程式會在驗證傳入 CSRs 的預設組態中公開 istiod 提供的相同服務。除了驗證之後，它會將請求轉換為 cert manager 支援的資源 （即與外部 CA 發行者 整合）。

每當有來自工作負載的 CSR 時，就會轉送至 istio-csr，其會向 ACM Private CA 請求憑證。AWS Private CA 發行者外掛程式會啟用 istio-csr 與 ACM Private CA 之間的通訊。 https://github.com/cert-manager/aws-privateca-issuerCert Manager 使用此外掛程式向 ACM Private CA 請求 TLS 憑證。發行者外掛程式將與 ACM Private CA 服務通訊，以請求工作負載的已簽署憑證。憑證簽署完成後，將傳回給 istio-csr，其將讀取已簽署的請求，並將其傳回給啟動 CSR 的工作負載。

使用 istio-csr 的 Istio 憑證簽署請求流程

image：：istio-csr-with-acm-private-ca.png【使用 istio-csr 簽署 Istio 憑證請求的流量】

Istio 搭配私有 CA 設定指示

1. 從遵循本節中的相同設定說明開始，完成下列操作：
2. 建立私有 CA
3. 安裝 cert-manager
4. 安裝發行者外掛程式
5. 設定許可並建立發行者。發行者代表 CA，並用於簽署istiod和網格工作負載憑證。它會與 ACM Private CA 通訊。
6. 建立 istio-system 命名空間。這是將部署 istiod certificate和其他 Istio 資源的位置。
7. 安裝使用 AWS Private CA 發行者外掛程式設定的 Istio CSR。您可以保留工作負載的憑證簽署請求，以確認它們獲得核准和簽署 (preserveCertificateRequests=true)。

```
helm install -n cert-manager cert-manager-istio-csr jetstack/cert-manager-istio-csr \
--set "app.certmanager.issuer.group=awspca.cert-manager.io" \
--set "app.certmanager.issuer.kind=AWSPCAClusterIssuer" \
--set "app.certmanager.issuer.name=<the-name-of-the-issuer-you-created>" \
--set "app.certmanager.preserveCertificateRequests=true" \
--set "app.server.maxCertificateDuration=48h" \
--set "app.tls.certificateDuration=24h" \
--set "app.tls.istiodCertificateDuration=24h" \
--set "app.tls.rootCAFile=/var/run/secrets/istio-csr/ca.pem" \
--set "volumeMounts[0].name=root-ca" \
--set "volumeMounts[0].mountPath=/var/run/secrets/istio-csr" \
--set "volumes[0].name=root-ca" \
--set "volumes[0].secret.secretName=istio-root-ca"
```

8. 使用自訂組態安裝 Istio，istiod以 取代 cert-manager istio-csr 作為網格的憑證提供者。您可以使用 Istio Operator 執行此程序。

```
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
metadata:
  name: istio
  namespace: istio-system
spec:
  profile: "demo"
  hub: gcr.io/istio-release
  values:
  global:
    # Change certificate provider to cert-manager istio agent for istio agent
```

```
    caAddress: cert-manager-istio-csr.cert-manager.svc:443
  components:
    pilot:
      k8s:
        env:
          # Disable istiod CA Sever functionality
        - name: ENABLE_CA_SERVER
          value: "false"
        overlays:
        - apiVersion: apps/v1
          kind: Deployment
          name: istiod
          patches:

            # Mount istiod serving and webhook certificate from Secret mount
          - path: spec.template.spec.containers.[name:discovery].args[7]
            value: "--tlsCertFile=/etc/cert-manager/tls/tls.crt"
          - path: spec.template.spec.containers.[name:discovery].args[8]
            value: "--tlsKeyFile=/etc/cert-manager/tls/tls.key"
          - path: spec.template.spec.containers.[name:discovery].args[9]
            value: "--caCertFile=/etc/cert-manager/ca/root-cert.pem"

          - path: spec.template.spec.containers.[name:discovery].volumeMounts[6]
            value:
              name: cert-manager
              mountPath: "/etc/cert-manager/tls"
              readOnly: true
          - path: spec.template.spec.containers.[name:discovery].volumeMounts[7]
            value:
              name: ca-root-cert
              mountPath: "/etc/cert-manager/ca"
              readOnly: true

          - path: spec.template.spec.volumes[6]
            value:
              name: cert-manager
              secret:
                secretName: istiod-tls
          - path: spec.template.spec.volumes[7]
            value:
              name: ca-root-cert
              configMap:
                defaultMode: 420
```

```
                name: istio-ca-root-cert
```

9. 部署您建立的上述自訂資源。

```
istioctl operator init
kubectl apply -f istio-custom-config.yaml
```

10現在，您可以將工作負載部署到 EKS 叢集中的網格，並強制執行 mTLS。

Istio 憑證簽署請求

image：：istio-csr-requests.png【Istio 憑證簽署請求】

工具和資源

- Amazon EKS 安全沉浸式研討會 - 網路安全
- 如何實作 cert-manager 和 ACM Private CA 外掛程式，以在 EKS 中啟用 TLS。
- 使用新的 AWS Load Balancer 控制器和 ACM Private CA 在 Amazon EKS 上設定end-to-end TLS 加密。
- GitHub 上的 Private CA Kubernetes cert-manager 外掛程式。
- Private CA Kubernetes cert-manager 外掛程式使用者指南。
- 如何使用 AWS Private Certificate Authority 短期憑證模式
- 輸出運算子 運算子和 DNS 外掛程式，無需通訊協定檢查即可控制來自叢集的輸出流量
- NeuVector by SUSE 開放原始碼、零信任容器安全平台提供政策網路規則、資料外洩防護 (DLP)、Web 應用程式防火牆 (WAF) 和網路威脅簽章。

資料加密和秘密管理

靜態加密

您可以搭配 Kubernetes 使用三種不同的 AWS 原生儲存選項：EBS、EFS 和 FSx for Lustre。這三個都使用服務受管金鑰或客戶主金鑰 (CMK) 提供靜態加密。對於 EBS，您可以使用樹狀內儲存驅動程式或 EBS CSI 驅動程式。兩者都包含用於加密磁碟區和提供 CMK 的參數。對於 EFS，您可以使用 EFS CSI 驅動程式，但與 EBS 不同，EFS CSI 驅動程式不支援動態佈建。如果您想要搭配 EKS 使用 EFS，在建立 PV 之前，您將需要為檔案系統佈建和設定靜態加密。如需 EFS 檔案加密的詳細資訊，請參閱加密靜態資料。除了提供靜態加密之外，EFS 和 FSx for Lustre 還包含加密傳

輸中資料的選項。FSx for Lustre 預設會執行此操作。對於 EFS，您可以透過將 `tls` 參數新增至 PV `mountOptions`中的 來新增傳輸加密，如本範例所示：

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: efs-pv
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: efs-sc
  mountOptions:
    - tls
  csi:
    driver: efs.csi.aws.com
    volumeHandle: <file_system_id>
```

FSx CSI 驅動程式支援動態佈建 Lustre 檔案系統。根據預設，它會使用服務受管金鑰加密資料，但您可以選擇提供您自己的 CMK，如本範例所示：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fsx-sc
provisioner: fsx.csi.aws.com
parameters:
  subnetId: subnet-056da83524edbe641
  securityGroupIds: sg-086f61ea73388fb6b
  deploymentType: PERSISTENT_1
  kmsKeyId: <kms_arn>
```

> **Important**
>
> 自 2020 年 5 月 28 日起，寫入 EKS Fargate Pod 中暫時性磁碟區的所有資料都會預設使用業界標準的 AES-256 密碼編譯演算法進行加密。不需要修改您的應用程式，因為 服務會無縫處理加密和解密。

加密靜態資料

加密靜態資料視為最佳實務。如果您不確定是否需要加密，請加密您的資料。

定期輪換您的 CMKs

設定 KMS 以自動輪換您的 CMKs。這將每年輪換您的金鑰一次，同時無限期儲存舊金鑰，以便仍然可以解密您的資料。如需詳細資訊，請參閱輪換客戶主金鑰

使用 EFS 存取點簡化對共用資料集的存取

如果您有具有不同 POSIX 檔案許可的共用資料集，或想要透過建立不同的掛載點來限制對共用檔案系統一部分的存取，請考慮使用 EFS 存取點。若要進一步了解如何使用存取點，請參閱 https：//https://docs.aws.amazon.com/efs/latest/ug/efs-access-points.html。今天，如果您想要使用存取點 (AP)，則需要在 PV `volumeHandle` 參數中參考 AP。

> **Important**
>
> 截至 2021 年 3 月 23 日，EFS CSI 驅動程式支援動態佈建 EFS 存取點。存取點是 EFS 檔案系統的應用程式特定進入點，可讓您更輕鬆地在多個 Pod 之間共用檔案系統。每個 EFS 檔案系統最多可有 120 PVs。如需其他資訊，請參閱介紹 Amazon EFS CSI 動態佈建。

秘密管理

Kubernetes 秘密用於存放敏感資訊，例如使用者憑證、密碼或 API 金鑰。它們會保留在以 base64 編碼的字串。在 EKS 上，已加密節點的 EBS 磁碟區會使用 EBS 加密進行加密。Pod 可以透過在 中參考秘密來擷取 Kubernetes 秘密物件`podSpec`。這些秘密可以映射到環境變數或掛載為磁碟區。如需建立秘密的詳細資訊，請參閱 https：//https://kubernetes.io/docs/concepts/configuration/secret/。

> **Warning**
>
> 秘密命名空間中的所有 Pod 都可以參考秘密命名空間中的秘密。

> **Warning**
>
> 節點授權方允許 Kubelet 讀取掛載到節點的所有秘密。

使用 AWS KMS 進行 Kubernetes 秘密的信封加密

這可讓您使用唯一的資料加密金鑰 (DEK) 來加密秘密。然後，DEK 會使用來自 AWS KMS 的金鑰加密金鑰 (KEK) 進行加密，該金鑰可依週期性排程自動輪換。使用適用於 Kubernetes 的 KMS 外掛程式，所有 Kubernetes 秘密都會以加密文字儲存在 中，而不是純文字，而且只能由 Kubernetes API 伺服器解密。如需其他詳細資訊，請參閱使用 EKS 加密提供者支援進行深度防禦

稽核 Kubernetes 秘密的使用

在 EKS 上，開啟稽核記錄並建立 CloudWatch 指標篩選條件和警示，以便在使用秘密時提醒您 （選用）。以下是 Kubernetes 稽核日誌 的指標篩選條件範例`{($.verb="get") && ($.objectRef.resource="secret")}`。您也可以搭配 CloudWatch Log Insights 使用下列查詢：

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| stats count(*) by objectRef.name as secret
| filter verb="get" and objectRef.resource="secrets"
```

上述查詢會顯示在特定時間範圍內存取秘密的次數。

```
fields @timestamp, @message
| sort @timestamp desc
| limit 100
| filter verb="get" and objectRef.resource="secrets"
| display objectRef.namespace, objectRef.name, user.username, responseStatus.code
```

此查詢會顯示秘密，以及嘗試存取秘密和回應代碼之使用者的命名空間和使用者名稱。

定期輪換您的秘密

Kubernetes 不會自動輪換秘密。如果您必須輪換秘密，請考慮使用外部秘密存放區，例如 Vault 或 AWS Secrets Manager。

使用個別命名空間做為隔離秘密與不同應用程式的方式

如果您有無法在命名空間中的應用程式之間共用的秘密，請為這些應用程式建立個別的命名空間。

使用磁碟區掛載而非環境變數

環境變數的值可能會無意中出現在日誌中。掛載為磁碟區的秘密會執行個體化為 tmpfs 磁碟區 (RAM 後端檔案系統），其會在刪除 Pod 時自動從節點移除。

使用外部秘密提供者

使用 Kubernetes 秘密有幾種可行的替代方案，包括 AWS Secrets Manager 和 Hashicorp 的保存庫。這些服務提供功能，例如精細存取控制、強式加密，以及 Kubernetes Secrets 無法使用的秘密自動輪換。Bitnami 的密封秘密是使用非對稱加密來建立「密封秘密」的另一種方法。公有金鑰用於加密秘密，而用於解密秘密的私有金鑰會保留在叢集中，可讓您將密封的秘密安全地存放在 Git 等來源控制系統中。如需詳細資訊，請參閱使用密封秘密在 Kubernetes 中管理秘密部署。

隨著外部秘密存放區的使用量不斷增加， 需要將它們與 Kubernetes 整合。Secret Store CSI 驅動程式是使用 CSI 驅動程式模型從外部秘密存放區擷取秘密的社群專案。目前，驅動程式支援 AWS Secrets Manager、Azure、Vault 和 GCP。AWS 提供者同時支援 AWS Secrets Manager 和 AWS 參數存放區。它也可以設定為在秘密過期時輪換秘密，並且可以將 AWS Secrets Manager 秘密同步到 Kubernetes Secrets。當您需要將秘密參考為環境變數，而不是從磁碟區讀取秘密時，秘密的同步會很有用。

> **Note**
>
> 當秘密存放區 CSI 驅動程式必須擷取秘密時，它會擔任指派給參考秘密之 Pod 的 IRSA 角色。您可以在此處找到此操作的程式碼。

如需 AWS Secrets & Configuration Provider (ASCP) 的其他資訊，請參閱下列資源：

- 如何使用 AWS Secrets 組態提供者搭配 Kubernetes Secret Store CSI 驅動程式
- 將 Secrets Manager 秘密與 Kubernetes Secrets Store CSI Driver 整合

external-secrets 是搭配 Kubernetes 使用外部秘密存放區的另一種方式。如同 CSI 驅動程式，外部秘密適用於各種不同的後端，包括 AWS Secrets Manager。差別在於，外部秘密不會從外部秘密存放區擷取秘密，而是將秘密從這些後端複製到 Kubernetes 做為秘密。這可讓您使用偏好的秘密存放區管理秘密，並以 Kubernetes 原生方式與秘密互動。

工具和資源

- Amazon EKS 安全浸入研討會 - 資料加密和秘密管理

執行期安全

執行期安全性可在容器執行時為您的容器提供主動保護。其目的是偵測和/或防止惡意活動在容器內發生。這可以透過與 Kubernetes 整合的 Linux 核心或核心延伸中的多種機制來實現，例如 Linux 功能、安全運算 (seccomp)、AppArmor 或 SELinux。還有 Amazon GuardDuty 和第三方工具等選項，可協助建立基準和偵測異常活動，並減少 Linux 核心機制的手動組態。

> **Important**
>
> Kubernetes 目前不提供將 seccomp、AppArmor 或 SELinux 設定檔載入節點的任何原生機制。它們必須在啟動時手動載入或安裝在節點上。這必須在 Pod 中參考它們之前完成，因為排程器不知道哪些節點具有設定檔。請參閱以下 Security Profiles Operator 等工具如何協助自動化將設定檔佈建至節點。

安全內容和內建的 Kubernetes 控制項

許多 Linux 執行期安全機制都與 Kubernetes 緊密整合，並且可以透過 Kubernetes 安全內容進行設定。其中一個選項是 `privileged`旗標，`false`預設為 ，如果啟用，基本上等同於主機上的根。在生產工作負載中啟用特權模式幾乎一律不適當，但有更多控制項可以視需要為容器提供更精細的權限。

Linux 功能

Linux 功能可讓您將特定功能授予 Pod 或容器，而無需提供根使用者的所有功能。範例包括 `CAP_NET_ADMIN`，其允許設定網路介面或防火牆，或 `CAP_SYS_TIME`，其允許操作系統時鐘。

Seccomp

透過安全運算 (seccomp)，您可以防止容器化應用程式對基礎主機作業系統的核心進行特定 syscall。雖然 Linux 作業系統有數百個系統呼叫，但 lion 的共用並不需要執行容器。透過限制容器可以進行的 syscall，您可以有效地減少應用程式的攻擊面。

Seccomp 的運作方式是攔截 syscall，並且只允許已列入允許清單的 傳遞。Docker 具有預設的 seccomp 設定檔，適用於大多數一般用途工作負載，而容器化等其他容器執行時間則提供相當的預設值。您可以將下列內容新增至 Pod 規格的 `securityContext`區段，將容器或 Pod 設定為使用容器執行期的預設 seccomp 設定檔：

```
securityContext:
  seccompProfile:
```

```
    type: RuntimeDefault
```

自 1.22 起（以 Alpha 表示，自 1.27 起穩定），上述`RuntimeDefault`可用於使用單一 kubelet 旗標的節點上的所有 Pod`--seccomp-default`。然後，只有在 中指定的設定檔`securityContext`才需要其他設定檔。

您也可以為需要額外權限的物件建立自己的設定檔。手動執行此操作可能會非常繁瑣，但有 Inspektor Gadget (在網路安全區段中也建議用於產生網路政策）和 Security Profiles Operator 等工具支援使用 eBPF 或 日誌等工具，將基準權限要求記錄為 seccomp 設定檔。Security Profiles Operator 進一步允許自動化將錄製的設定檔部署到節點，以供 Pod 和容器使用。

AppArmor 和 SELinux

AppArmor 和 SELinux 稱為強制性存取控制或 MAC 系統。其概念與 seccomp 相似，但具有不同的 APIs和功能，允許特定 檔案系統路徑或網路連接埠的存取控制。這些工具的支援取決於 Linux 發行版本，Debian/Ubuntu 支援 AppArmor 和 RHEL/CentOS/Bottlerocket/Amazon Linux 2023 支援 SELinux。另請參閱基礎設施安全章節，以進一步討論 SELinux。

AppArmor 和 SELinux 都與 Kubernetes 整合，但從 Kubernetes 1.28 AppArmor 設定檔必須透過註釋指定，而 SELinux 標籤可以透過安全內容上的 SELinuxOptions 欄位直接設定。

如同 seccomp 描述檔，上述安全描述檔運算子可協助將描述檔部署到叢集中的節點。（未來，專案也旨在產生 AppArmor 和 SELinux 的設定檔，如同 seccomp 一樣。）

建議

使用 Amazon GuardDuty 來監控執行時間並偵測對 EKS 環境的威脅

如果您目前沒有持續監控 EKS 執行時間和分析 EKS 稽核日誌，以及掃描惡意軟體和其他可疑活動的解決方案，Amazon 強烈建議希望以簡單、快速、安全、可擴展且經濟實惠的一鍵式方式保護其 AWS 環境的客戶使用 Amazon GuardDuty。Amazon GuardDuty 是一種安全監控服務，可分析和處理基礎資料來源，例如 AWS CloudTrail 管理事件、AWS CloudTrail 事件日誌、VPC 流程日誌（來自 Amazon EC2 執行個體）、Kubernetes 稽核日誌和 DNS 日誌。它還包括 EKS 執行期監控。它使用持續更新的威脅情報摘要，例如惡意 IP 地址和網域的清單，以及機器學習，來識別 AWS 環境中非預期、可能未經授權的惡意活動。這可能包括權限提升、使用公開的登入資料，或與惡意 IP 地址、網域的通訊、Amazon EC2 執行個體和 EKS 容器工作負載上存在惡意軟體，或探索可疑 API 活動等問題。GuardDuty 透過產生您可以在 GuardDuty 主控台或透過 Amazon EventBridge 檢視的安全性調查結果，通知您 AWS 環境的狀態。GuardDuty 也支援您將問題清單匯出至 Amazon Simple Storage Service (S3) 儲存貯體，並與 AWS Security Hub 和 Detective 等其他服務整合。

觀看此 AWS Online Tech Talk「使用 Amazon GuardDuty 增強 Amazon EKS 的威脅偵測 - AWS Online Tech Talks」，了解如何在幾分鐘內step-by-step啟用這些額外的 EKS 安全功能。

選用：使用第三方解決方案進行執行期監控

如果您不熟悉 Linux 安全性，建立和管理 seccomp 和 Apparmor 設定檔可能會很困難。如果您沒有時間精通，請考慮使用第三方商業解決方案。其中許多都已超越 Apparmor 和 seccomp 等靜態設定檔，並開始使用機器學習來封鎖或提醒可疑活動。您可以在以下工具區段中找到其中一些解決方案。您可以在 AWS Marketplace for Containers 中找到其他選項。

在撰寫 seccomp 政策之前，請考慮新增/捨棄 Linux 功能

功能涉及 syscall 可存取的核心函數中的各種檢查。如果檢查失敗，syscall 通常會傳回錯誤。檢查可以在特定 syscall 開始時完成，或在可透過多個不同 syscall 連線的區域中更深入的核心（例如寫入特定特權檔案）。另一方面，Seccomp 是一種 syscall 篩選條件，在執行之前會套用至所有 syscall。程序可以設定篩選條件，允許他們撤銷執行特定 syscall 的權利，或特定 syscall 的特定引數。

使用 seccomp 之前，請考慮新增/移除 Linux 功能是否提供您所需的控制。如需詳細資訊，請參閱設定容器的功能。

了解您是否可以使用 Pod 安全政策 (PSPs) 達成目標

Pod 安全政策提供許多不同的方式來改善您的安全狀態，而不會帶來過度的複雜性。探索 PSPs中可用的選項，然後再進入建置 seccomp 和 Apparmor 設定檔。

> **Warning**
>
> 從 Kubernetes 1.25 開始，已移除 PSPs並以 Pod 安全許可控制器取代 PSP。存在的第三方替代方案包括 OPA/Gatekeeper 和 Kyverno。用於實作 PSPs 中常見政策的 Gatekeeper 限制和限制範本集合可從 GitHub 上的 Gatekeeper 程式庫儲存庫提取。您可以在 Kyverno 政策程式庫中找到許多 PSPs 的替換，包括完整的 Pod 安全標準集合。

工具與資源

- 開始之前您應該知道的 7 件事
- AppArmor 載入器
- 使用設定檔設定節點

- Security Profiles Operator 是一種 Kubernetes 增強功能，旨在讓使用者更輕鬆地在 Kubernetes 叢集中使用 SELinux、seccomp 和 AppArmor。它提供從執行中的工作負載產生設定檔，以及將設定檔載入 Kubernetes 節點以用於 Pod 的功能。
- Inspektor Gadget 允許在 Kubernetes 上檢查、追蹤和分析執行時間行為的許多層面，包括協助產生 seccomp 設定檔。
- Aqua
- Qualys
- Stackrox
- Sysdig 安全
- Prisma
- NeuVector by SUSE 開放原始碼、零信任容器安全平台提供程序設定檔規則和檔案存取規則。

保護基礎設施（主機）

由於保護您的容器映像非常重要，因此保護執行它們的基礎設施也同樣重要。本節會探索不同方式，以降低直接針對主機啟動之攻擊的風險。這些準則應與執行期安全性一節中概述的準則搭配使用。

建議

使用針對執行中容器最佳化的作業系統

請考慮使用 Flatcar Linux、Project Atomic、RancherOS 和 Bottlerocket，這是 AWS 專為執行 Linux 容器而設計的特殊用途作業系統。它包含減少的攻擊面、開機時驗證的磁碟映像，以及使用 SELinux 強制執行的許可界限。

或者，為您的 Kubernetes 工作者節點使用 EKS 最佳化 AMI。EKS 最佳化 AMI 會定期發行，其中包含執行容器化工作負載所需的一組最少作業系統套件和二進位檔。

如需範例組態指令碼，請參閱 Amazon EKS AMI RHEL Build Specification，該指令碼可用於建置使用 Hashicorp Packer 在 Red Hat Enterprise Linux 上執行的自訂 Amazon EKS AMI。您可以進一步利用此指令碼來建置符合 STIG 規範的 EKS 自訂 AMIs。

讓您的工作者節點作業系統保持最新狀態

無論您使用的是 Bottlerocket 等容器最佳化主機作業系統還是更大的主機作業系統，但仍是極簡的 Amazon Machine Image，例如 EKS 最佳化 AMIs，最佳實務是將這些主機作業系統映像與最新的安全修補程式保持最新狀態。

對於 EKS 最佳化 AMIs，請定期檢查 CHANGELOG 和/或版本備註管道，並自動將更新的工作者節點映像推展到您的叢集。

將您的基礎設施視為不可變，並自動取代您的工作者節點

新的修補程式或更新可用時，請取代您的工作者，而不是執行就地升級。這可以用幾種方式進行。您可以在依序封鎖和耗盡節點時，使用最新的 AMI 將執行個體新增至現有的自動擴展群組，直到群組中的所有節點都取代為最新的 AMI 為止。或者，您可以將執行個體新增至新的節點群組，同時依序從舊節點群組封鎖和耗盡節點，直到取代所有節點為止。EKS 受管節點群組使用第一種方法，並在有新 AMI 可用時，在主控台中顯示訊息來升級工作者。 `eksctl` 也有一種機制，可用來建立具有最新 AMI 的節點群組，並在執行個體終止之前從節點群組正常封鎖和耗盡 Pod。如果您決定使用不同的方法來取代工作者節點，強烈建議您自動化程序以將人為監督降至最低，因為當新的更新/修補程式發佈時以及控制平面升級時，您可能需要定期取代工作者。

透過 EKS Fargate，AWS 會在更新可用時自動更新基礎基礎設施。通常可以順暢地完成此操作，但有時候更新可能會導致您的 Pod 重新排程。因此，我們建議您在以 Fargate Pod 身分執行應用程式時，使用多個複本建立部署。

定期執行 kube-bench，以驗證是否符合 Kubernetes 的 CIS 基準

kube-bench 是 Aqua 的開放原始碼專案，可根據 Kubernetes 的 CIS 基準評估您的叢集。基準描述保護未受管 Kubernetes 叢集的最佳實務。CIS Kubernetes 基準包含控制平面和資料平面。由於 Amazon EKS 提供全受管控制平面，因此並非所有來自 CIS Kubernetes Benchmark 的建議都適用。為了確保此範圍反映 Amazon EKS 的實作方式，AWS 建立了 CIS Amazon EKS 基準。EKS 基準從 CIS Kubernetes Benchmark 繼承，並具有 EKS 叢集特定組態考量的其他社群輸入。

針對 EKS 叢集執行 kube-bench 時，請遵循 Aqua Security 的這些指示。如需詳細資訊，請參閱簡介 CIS Amazon EKS 基準。

將對工作者節點的存取降到最低

當您需要遠端存取主機時，請使用 SSM Session Manager，而不是啟用 SSH 存取。與可能遺失、複製或共用的 SSH 金鑰不同，Session Manager 可讓您使用 IAM 控制對 EC2 執行個體的存取。此外，它還提供在執行個體上執行之命令的稽核線索和日誌。

自 2020 年 8 月 19 日起，受管節點群組支援自訂 AMIs 和 EC2 啟動範本。這可讓您將 SSM 代理程式嵌入 AMI，或在工作者節點啟動時安裝它。如果您不想修改最佳化 AMI 或 ASG 的啟動範本，您可以使用 DaemonSet 安裝 SSM 代理程式，如本範例所示。

SSM 型 SSH 存取的最低 IAM 政策

如果您只想避免 SSH 存取，AmazonSSMManagedInstanceCoreAWS 受管政策包含 SSM Session Manager / SSM RunCommand 不需要的許多許可。需要特別注意ssm:GetParameter(s)的是允許角色存取參數存放區中所有參數的*許可 （ 包括已設定 AWS 受管 KMS 金鑰的 SecureStrings)。

下列 IAM 政策包含透過 SSM Systems Manager 啟用節點存取的最小許可集。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "EnableAccessViaSSMSessionManager",
      "Effect": "Allow",
      "Action": [
        "ssmmessages:OpenDataChannel",
        "ssmmessages:OpenControlChannel",
        "ssmmessages:CreateDataChannel",
        "ssmmessages:CreateControlChannel",
        "ssm:UpdateInstanceInformation"
      ],
      "Resource": "*"
    },
    {
      "Sid": "EnableSSMRunCommand",
      "Effect": "Allow",
      "Action": [
        "ssm:UpdateInstanceInformation",
        "ec2messages:SendReply",
        "ec2messages:GetMessages",
        "ec2messages:GetEndpoint",
        "ec2messages:FailMessage",
        "ec2messages:DeleteMessage",
        "ec2messages:AcknowledgeMessage"
      ],
      "Resource": "*"
    }
  ]
}
```

設定此政策並安裝 Session Manager 外掛程式後，您就可以執行

```
aws ssm start-session --target [INSTANCE_ID_OF_EKS_NODE]
```

以存取節點。

> **Note**
>
> 您也可以考慮新增許可，以啟用 Session Manager 記錄。

將工作者部署到私有子網路

透過將工作者部署到私有子網路，您可以將他們對攻擊經常產生的網際網路的暴露降至最低。從 2020 年 4 月 22 日開始，受管節點群組中節點的公有 IP 地址指派將由其部署所在的子網路控制。在此之前，受管節點群組中的節點會自動指派公有 IP。如果您選擇將工作者節點部署到公有子網路，請實作限制性 AWS 安全群組規則來限制其暴露。

執行 Amazon Inspector 來評估主機的暴露、漏洞和與最佳實務的偏差

您可以使用 Amazon Inspector 來檢查節點的意外網路存取，以及基礎 Amazon EC2 執行個體上的漏洞。

只有在安裝並啟用 Amazon EC2 Systems Manager (SSM) 代理程式時，Amazon Inspector 才能為您的 Amazon EC2 執行個體提供常見的漏洞和暴露 (CVE) 資料。 Amazon EC2 Systems Manager 此代理程式預先安裝在數個 Amazon Machine Image AMIs) 上，包括 EKS 最佳化的 Amazon Linux AMIs。無論 SSM 代理程式狀態為何，都會掃描所有 Amazon EC2 執行個體是否有網路連線能力問題。如需設定 Amazon EC2 掃描的詳細資訊，請參閱掃描 Amazon EC2 執行個體。

> **Important**
>
> Inspector 無法在用來執行 Fargate Pod 的基礎設施上執行。

替代方案

執行 SELinux

> **Note**
>
> 適用於 Red Hat Enterprise Linux (RHEL)、CentOS、 Bottlerocket 和 Amazon Linux 2023

SELinux 提供額外的安全層，讓容器彼此和主機保持隔離。SELinux 可讓管理員針對每個使用者、應用程式、程序和檔案強制執行強制性存取控制 (MAC)。將其視為根據一組標籤，將可對 執行的操作限制為特定資源的後端。在 EKS 上，SELinux 可用來防止容器存取彼此的資源。

Container SELinux 政策在 container-selinux 套件中定義。Docker CE 需要此套件 （以及其相依性），以便 Docker （或其他容器執行時間） 建立的程序和檔案在有限的系統存取下執行。容器會利用的別名`container_t`標籤`svirt_lxc_net_t`。這些政策可有效防止容器存取主機的某些功能。

當您為 Docker 設定 SELinux 時，Docker 會自動`container_t`將工作負載標記為類型，並為每個容器提供唯一的 MCS 層級。這會將容器彼此隔離。如果您需要更寬鬆的限制，您可以在 SElinux 中建立自己的設定檔，將容器許可授予檔案系統的特定區域。這類似於 中的 PSPs，您可以為不同的容器/Pod 建立不同的設定檔。例如，您可以為具有一組限制性控制項的一般工作負載建立設定檔，並為需要特殊權限存取的物件建立設定檔。

SELinux for Containers 有一組選項，可設定為修改預設限制。您可以根據您的需求啟用或停用下列 SELinux 布林值：

Boolean	預設	說明
`container_connect_any`	`off`	允許容器存取主機上的特權連接埠。例如，如果您的容器需要將連接埠映射到主機上的 443 或 80。
`container_manage_cgroup`	`off`	允許容器管理 cgroup 組態。例如，執行 systemd 的容器將需要啟用此功能。
`container_use_cephfs`	`off`	允許容器使用 ceph 檔案系統。

根據預設，允許容器在 下讀取/執行，`/usr`並從 讀取大部分內容`/etc`。`/var/lib/docker` 和 下的檔案`/var/lib/containers`具有標籤 `container_var_lib_t`。若要檢視完整的預設清單，標籤會看到 container.fc 檔案。

```
docker container run -it \
  -v /var/lib/docker/image/overlay2/repositories.json:/host/repositories.json \
  centos:7 cat /host/repositories.json
```

```
# cat: /host/repositories.json: Permission denied

docker container run -it \
  -v /etc/passwd:/host/etc/passwd \
  centos:7 cat /host/etc/passwd
# cat: /host/etc/passwd: Permission denied
```

標示container_file_t為 的檔案是容器可寫入的唯一檔案。如果您希望磁碟區掛載可寫入，則需要:Z在結尾指定 :z或 。

- :z 會重新標記檔案，讓容器可以讀取/寫入
- :Z 會重新標記檔案，以便只有容器可以讀取/寫入

```
ls -Z /var/lib/misc
# -rw-r--r--. root root system_u:object_r:var_lib_t:s0   postfix.aliasesdb-stamp

docker container run -it \
  -v /var/lib/misc:/host/var/lib/misc:z \
  centos:7 echo "Relabeled!"

ls -Z /var/lib/misc
#-rw-r--r--. root root system_u:object_r:container_file_t:s0 postfix.aliasesdb-stamp
```

```
docker container run -it \
  -v /var/log:/host/var/log:Z \
  fluentbit:latest
```

在 Kubernetes 中，重新標記略有不同。您可以指定自訂 MCS 標籤來執行 Pod，而不是讓 Docker 自動重新標記檔案。支援重新標記的磁碟區會自動重新標記，以便存取。具有相符 MCS 標籤的 Pod 將能夠存取磁碟區。如果您需要嚴格的隔離，請為每個 Pod 設定不同的 MCS 標籤。

```
securityContext:
  seLinuxOptions:
    # Provide a unique MCS label per container
    # You can specify user, role, and type also
    # enforcement based on type and level (svert)
    level: s0:c144:c154
```

在此範例中，s0:c144:c154對應至指派給允許容器存取之檔案的 MCS 標籤。

在 EKS 上，您可以建立允許特殊權限容器執行的政策，例如 FluentD，並建立 SELinux 政策，以允許它從主機上的 /var/log 讀取，而不需要重新標記主機目錄。具有相同標籤的 Pod 將能夠存取相同的主機磁碟區。

我們已針對已在 CentOS 7 和 RHEL 7 上設定 SELinux 的 Amazon EKS 實作範例 AMIs。這些 AMIs的開發目的是示範符合高度管制客戶要求的範例實作。

> ⚠ Warning
>
> SELinux 會忽略類型未受限的容器。

工具和資源

- SELinux Kubernetes RBAC 和內部部署應用程式的運送安全政策
- Kubernetes 的反覆強化
- Audit2Allow
- SEAlert
- 使用 Udica 為容器產生 SELinux 政策描述一種工具，可查看 Linux 功能、連接埠和掛載點的容器規格檔案，並產生一組 SELinux 規則，允許容器正常執行
- AMI 強化手冊，用於強化作業系統以符合不同的法規要求
- Keiko Upgrade Manager 是來自 Intuit 的開放原始碼專案，可協調工作者節點的輪換。
- Sysdig 安全
- eksctl

合規

合規是 AWS 與其服務消費者共同的責任。一般而言，AWS 負責「雲端安全」，而其使用者負責「雲端安全」。說明 AWS 及其使用者所負責內容的行會因服務而異。例如，透過 Fargate，AWS 負責管理其資料中心、硬體、虛擬基礎設施 (Amazon EC2) 和容器執行期 (Docker) 的實體安全。Fargate 的使用者負責保護容器映像及其應用程式的安全。在執行必須遵循合規標準的工作負載時，了解誰負責處理什麼是重要的考量事項。

下表顯示不同容器服務符合的合規計劃。

合規計劃	Amazon ECS Orchestrator	Amazon EKS Orchestrator	ECS Fargate	Amazon ECR
PCI DSS 第 1 級	1	1	1	1
HIPAA 合格服務	1	1	1	1
SOC I	1	1	1	1
SOC II	1	1	1	1
SOC III	1	1	1	1
ISO 27001：2013	1	1	1	1
ISO 9001：2015	1	1	1	1
ISO 27017：2015	1	1	1	1
ISO 27018：2019	1	1	1	1
IRAP	1	1	1	1
FedRAMP Moderate (East/West)	1	1	0	1
FedRAMP High (GovCloud)	1	1	0	1
DOD CC SRG	1	DISA Review (IL5)	0	1
HIPAA BAA	1	1	1	1
MTCS	1	1	0	1
C5	1	1	0	1
K-ISMS	1	1	0	1
ENS 高級	1	1	0	1

合規計劃	Amazon ECS Orchestrator	Amazon EKS Orchestrator	ECS Fargate	Amazon ECR
OSPAR	1	1	0	1
HITRUST CSF	1	1	1	1

合規狀態會隨著時間而變更。如需最新狀態，請務必參閱 https：//https://aws.amazon.com/compliance/services-in-scope/。

如需雲端認證模型和最佳實務的詳細資訊，請參閱 AWS 白皮書，安全雲端採用的認證模型

向左移

向左轉移的概念包括在軟體開發生命週期中稍早發現政策違規和錯誤。從安全角度來看，這可能非常有益。例如，開發人員可以在其應用程式部署到叢集之前，修正其組態的問題。像先前這樣的錯誤有助於防止部署違反政策的組態。

政策即程式碼

政策可視為一組管理行為的規則，即允許的行為或禁止 的行為。例如，您可能有一個政策，指出所有 Dockerfiles 都應包含 USER 指令，導致容器以非根使用者身分執行。做為文件，這類政策可能很難探索和強制執行。它也可能隨著您的需求變更而過時。透過政策即程式碼 (PaC) 解決方案，您可以自動化安全、合規和隱私權控制，以偵測、預防、減少和對抗已知和持久性的威脅。此外，它們為您提供機制來編纂您的政策，並在您執行其他程式碼成品時對其進行管理。這種方法的好處是您可以重複使用 DevOps 和 GitOps 策略，以管理和一致地在 Kubernetes 叢集的機群間套用政策。如需 PaC 選項和 PSPs 未來的相關資訊，請參閱 Pod Security。

在管道中使用policy-as-code工具，在部署之前偵測違規

- OPA 是一種開放原始碼政策引擎，屬於 CNCF 的一部分。它用於做出政策決策，並且可以執行各種不同的方式，例如 做為語言程式庫或服務。OPA 政策是以稱為 Rego 的網域特定語言 (DSL) 撰寫。雖然它通常作為 Gatekeeper 專案的 Kubernetes 動態許可控制器的一部分執行，但 OPA 也可以納入您的 CI/CD 管道。這可讓開發人員在發行週期之前取得有關其組態的意見回饋，進而協助他們在進入生產環境之前解決問題。您可以在此專案的 GitHub 儲存庫中找到常見 OPA 政策的集合。
- Conftest 建立在 OPA 之上，可提供以開發人員為重心的測試 Kubernetes 組態體驗。
- Kyverno 是專為 Kubernetes 設計的政策引擎。使用 Kyverno 時，政策會做為 Kubernetes 資源進行管理，而且撰寫政策時不需要使用新語言。這允許使用熟悉的工具，例如 kubectl、git 和 kustomize

來管理政策。Kyverno 政策可以驗證、變更和產生 Kubernetes 資源，並確保 OCI 映像供應鏈安全性。Kyverno CLI 可用來測試政策，並驗證做為 CI/CD 管道一部分的資源。您可以在 Kyverno 網站上找到所有 Kyverno 社群政策，如需使用 Kyverno CLI 在管道中撰寫測試的範例，請參閱政策儲存庫。

工具和資源

- Amazon EKS 安全沉浸研討會 - 法規合規
- kube-bench
- docker-bench-security
- AWS Inspector
- Kubernetes 安全審查 Kubernetes 1.13.4 的第三方安全評估 (2019)
- NeuVector by SUSE 開放原始碼、零信任容器安全平台，提供合規報告和自訂合規檢查

事件回應和鑑識

您快速對事件做出反應的能力，有助於將違規所造成的損害降至最低。擁有可警告您可疑行為的可靠提醒系統，是良好事件回應計畫的第一步。當事件發生時，您必須快速決定是否銷毀和取代受影響的容器，或隔離和檢查容器。如果您選擇隔離容器做為鑑識調查和根本原因分析的一部分，則應遵循下列一組活動：

事件回應計劃範例

識別違規的 Pod 和工作者節點

您的第一個動作應該是隔離損壞。首先識別違規發生的位置，並將該 Pod 及其節點與基礎設施的其他部分隔離。

使用工作負載名稱識別違規的 Pod 和工作者節點

如果您知道違規 Pod 的名稱和命名空間，您可以識別執行 Pod 的工作者節點，如下所示：

```
kubectl get pods <name> --namespace <namespace> -o=jsonpath='{.spec.nodeName}{"\n"}'
```

如果部署等工作負載資源遭到入侵，則屬於工作負載資源的所有 Pod 都可能遭到入侵。使用下列命令列出工作負載資源的所有 Pod 及其執行的節點：

```
selector=$(kubectl get deployments <name> \
 --namespace <namespace> -o json | jq -j \
'.spec.selector.matchLabels | to_entries | .[] | "\(.key)=\(.value)"')

kubectl get pods --namespace <namespace> --selector=$selector \
-o json | jq -r '.items[] | "\(.metadata.name) \(.spec.nodeName)"'
```

上述命令適用於 部署。您可以針對複本、狀態集等其他工作負載資源執行相同的命令。

使用服務帳戶名稱識別違規的 Pod 和工作者節點

在某些情況下，您可能會發現服務帳戶遭到入侵。使用已識別服務帳戶的 Pod 可能會遭到入侵。您可以使用服務帳戶和其執行的節點，使用下列命令來識別所有 Pod：

```
kubectl get pods -o json --namespace <namespace> | \
    jq -r '.items[] |
    select(.spec.serviceAccount == "<service account name>") |
    "\(.metadata.name) \(.spec.nodeName)"'
```

識別具有易受攻擊或遭到入侵的映像和工作者節點的 Pod

在某些情況下，您可能會發現叢集上 Pod 中使用的容器映像是惡意或洩露的。如果發現容器映像包含惡意軟體，則容器映像是惡意或洩露的，是已知的不良映像，或具有已被利用的 CVE。您應該考慮所有使用容器映像的 Pod 都遭到入侵。您可以使用正在執行的映像和節點，使用以下命令來識別 Pod：

```
IMAGE=<Name of the malicious/compromised image>

kubectl get pods -o json --all-namespaces | \
    jq -r --arg image "$IMAGE" '.items[] |
    select(.spec.containers[] | .image == $image) |
    "\(.metadata.name) \(.metadata.namespace) \(.spec.nodeName)"'
```

透過建立網路政策來隔離 Pod，該政策會拒絕所有傳入和傳出至 Pod 的流量

拒絕所有流量規則可透過切斷與 Pod 的所有連線，協助停止已進行中的攻擊。下列網路政策將套用至標籤為 的 Podapp=web。

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
```

```
spec:
  podSelector:
    matchLabels:
      app: web
  policyTypes:
  - Ingress
  - Egress
```

> **Important**
>
> 如果攻擊者已取得基礎主機的存取權，則網路政策可能無效。如果您懷疑 已發生，您可以使用 AWS 安全群組來隔離遭入侵的主機與其他主機。變更主機的安全群組時，請注意，這會影響在該主機上執行的所有容器。

視需要撤銷指派給 Pod 或工作者節點的臨時安全登入資料

如果工作者節點已指派 IAM 角色，允許 Pod 存取其他 AWS 資源，請從執行個體中移除這些角色，以防止進一步的攻擊傷害。同樣地，如果已為 Pod 指派 IAM 角色，請評估您是否可以安全地從該角色中移除 IAM 政策，而不會影響其他工作負載。

繫結工作者節點

透過隔離受影響的工作者節點，您可以通知排程器，以避免將 Pod 排程到受影響的節點。這可讓您移除鑑識研究的節點，而不會中斷其他工作負載。

> **Note**
>
> 本指南不適用於 Fargate，其中每個 Fargate Pod 在自己的沙盒環境中執行。套用拒絕所有傳入和傳出流量的網路政策，取代封鎖，以封鎖受影響的 Fargate Pod。

在受影響的工作者節點上啟用終止保護

攻擊者可能會嘗試透過終止受影響的節點來清除其行為錯誤。啟用終止保護可以防止這種情況發生。執行個體縮減保護會保護節點免於縮減事件。

> **Warning**
>
> 您無法在 Spot 執行個體上啟用終止保護。

使用標籤標記違規的 Pod/Node，指出它是主動調查的一部分

這將作為警告，讓叢集管理員在調查完成之前不要竄改受影響的 Pod/節點。

擷取工作者節點上的揮發性成品

- 擷取作業系統記憶體。這將擷取每個容器的 Docker 協助程式（或其他容器執行時間）及其子程序。這可以使用 LiME 和 Volatility 等工具，或透過在上面建置的 Amazon EC2 專用 Automated Forensics Orchestrator 等高階工具來完成。
- 執行執行中程序和開放連接埠的 netstat 樹傾印。這將擷取每個容器的 docker 協助程式及其子程序。
- 在修改證據之前，執行 命令來儲存容器層級狀態。您可以使用容器執行時間的功能來擷取目前執行中容器的相關資訊。例如，使用 Containerd，您可以執行下列動作：
  - `crictl ps` 適用於執行中的程序。
  - `crictl logs CONTAINER` 適用於常駐程式層級的保留日誌。

    使用 nerdctl CLI 取代 `docker`（例如 `nerdctl inspect`) 以容器化 來達成相同的目標。某些額外的命令會根據容器執行時間提供。例如，Docker `docker diff` 必須查看容器檔案系統的變更，或儲存所有容器狀態`docker checkpoint`，包括揮發性記憶體 (RAM)。如需與容器或 CRI-O 執行時間類似功能的討論，請參閱此 Kubernetes 部落格文章。
- 暫停容器以進行鑑識擷取。
- 快照執行個體的 EBS 磁碟區。

重新部署遭入侵的 Pod 或工作負載資源

收集資料進行鑑識分析之後，您就可以重新部署遭入侵的 Pod 或工作負載資源。

首先針對遭到入侵的漏洞推出修正，並啟動新的替換 Pod。然後刪除易受攻擊的 Pod。

如果易受攻擊的 Pod 是由更高層級的 Kubernetes 工作負載資源（例如部署或 DaemonSet) 管理，則刪除它們會排程新的 Pod。因此，易受攻擊的 Pod 將再次啟動。在這種情況下，您應該在修正漏洞之後部署新的替換工作負載資源。然後，您應該刪除易受攻擊的工作負載。

建議

檢閱 AWS 安全事件回應白皮書

雖然本節提供簡短的概觀，以及處理可疑安全漏洞的一些建議，但白皮書 AWS 安全事件回應會詳細介紹主題。

練習安全遊戲日

將您的安全從業人員分成 2 個團隊：紅色和藍色。紅隊將專注於探測不同系統是否有漏洞，而藍隊將負責防禦漏洞。如果您沒有足夠的安全從業人員來建立不同的團隊，請考慮雇用熟悉 Kubernetes 漏洞的外部實體。

Kubesploit 是 CyberArk 的滲透測試架構，可用來執行遊戲日。與其他掃描叢集是否有漏洞的工具不同，kubesploit 模擬真實世界的攻擊。這讓您的藍隊有機會練習其對攻擊的回應，並衡量其有效性。

針對您的叢集執行滲透測試

定期攻擊您自己的叢集可協助您發現漏洞和設定錯誤。開始使用之前，請先遵循滲透測試準則，再對您的叢集進行測試。

工具和資源

- kube-hunter，適用於 Kubernetes 的滲透測試工具。
- Gremlin 是一種混沌的工程工具組，可用來模擬針對應用程式和基礎設施的攻擊。
- 攻擊和防禦 Kubernetes 安裝
- kubesploit
- NeuVector by SUSE 開放原始碼、零信任容器安全平台，提供漏洞和風險報告以及安全事件通知
- 進階持久性威脅
- Kubernetes 實務攻擊與防禦
- 利用 RBAC 許可入侵 Kubernetes 叢集

映像安全性

您應該將容器映像視為對抗攻擊的第一道防線。不安全、構造不良的映像可允許攻擊者逸出容器的邊界並取得主機的存取。一旦在主機上，攻擊者就可以存取敏感資訊，或在叢集內或透過您的 AWS 帳戶橫向移動。下列最佳實務將有助於降低發生這種情況的風險。

建議

建立最小影像

首先從容器映像中刪除所有無關的二進位檔案。如果您使用來自 Dockerhub 的不熟悉映像，請使用 Dive 之類的應用程式來檢查映像，該應用程式可以向您顯示每個容器層的內容。移除具有 SETUID 和 SETGID 位元的所有二進位檔，因為它們可用於提升權限，並考慮移除可用於惡意目的的所有 shell 和公用程式，例如 nc 和 curl。您可以使用下列命令找到具有 SETUID 和 SETGID 位元的檔案：

```
find / -perm /6000 -type f -exec ls -ld {} \;
```

若要從這些檔案移除特殊許可，請將下列指令新增至您的容器映像：

```
RUN find / -xdev -perm /6000 -type f -exec chmod a-s {} \; || true
```

以共通方式來說，這稱為取消分散您的映像。

使用多階段建置

使用多階段建置是一種建立最小影像的方法。多階段建置通常用於自動化持續整合週期的一部分。例如，多階段組建可用來固定原始程式碼或執行靜態程式碼分析。這讓開發人員有機會取得近乎立即的意見回饋，而不是等待管道執行。從安全角度來看，多階段建置很有吸引力，因為它們可讓您將推送至容器登錄檔的最終映像大小降至最低。沒有構建工具和其他無關二進位檔案的容器映像透過減少映像的受攻擊面來改善您的安全狀態。如需多階段建置的其他資訊，請參閱 Docker 的多階段建置文件。

為您的容器映像建立軟體物料清單 SBOMs)

「軟體物料清單」(SBOM) 是構成容器映像之軟體成品的巢狀清查。SBOM 是軟體安全和軟體供應鏈風險管理的關鍵建置區塊。在中央儲存庫中產生、儲存 SBOMS 和掃描 SBOMs是否有漏洞，有助於解決下列問題：

- 可見性：了解哪些元件構成您的容器映像。存放在中央儲存庫中可讓 SBOMs 隨時接受稽核和掃描，甚至在部署後偵測和回應新的漏洞，例如零時差漏洞。
- 實證驗證：確保成品源自何處及如何源自的現有假設為 true，以及成品或其隨附的中繼資料在建置或交付程序期間並未遭到竄改。
- 可信度：確保可信任指定的成品及其內容來執行其聲稱要執行的動作，即 適用於特定用途。這涉及判斷程式碼是否可安全執行，並針對與執行程式碼相關的風險做出明智決策。透過建立已證明的管道

執行報告以及已證明的 SBOM 和已證明的 CVE 掃描報告來確保可信度，以確保映像的消費者透過安全方式（管道）使用安全元件建立此映像。

- 相依性信任驗證：遞迴檢查成品的相依性樹狀結構是否有可信性和成品的來源。SBOMs中的漂移有助於偵測惡意活動，包括未經授權、不受信任的相依性、滲透嘗試。

下列工具可用於產生 SBOM：

- Amazon Inspector 可用來建立和匯出 SBOMs。
- 來自 Anchore 的 Syft 也可以用於產生 SBOM。為了更快地掃描漏洞，為容器映像產生的 SBOM 可以用作要掃描的輸入。然後，SBOM 和掃描報告會經過證明並附加至映像，再將映像推送至中央 OCI 儲存庫，例如 Amazon ECR，以供檢閱和稽核之用。

請參閱 CNCF 軟體供應鏈最佳實務指南，進一步了解如何保護您的軟體供應鏈。

定期掃描映像是否有漏洞

如同其虛擬機器對等，容器映像可以包含具有漏洞的二進位檔和應用程式程式庫，或隨著時間開發漏洞。防止漏洞攻擊的最佳方法是使用映像掃描器定期掃描映像。存放在 Amazon ECR 中的影像可以在推送或隨需時掃描 (24 小時期間內掃描一次）。ECR 目前支援兩種類型的掃描 - Basic 和 Enhanced。基本掃描會免費利用 Clair 開放原始碼映像掃描解決方案。增強型掃描使用 Amazon Inspector 提供自動持續掃描，以支付額外費用。掃描影像後，結果會記錄到 EventBridge 中 ECR 的事件串流。您也可以在 ECR 主控台中查看掃描的結果。應刪除或重建具有 HIGH 或 CRITICAL 漏洞的影像。如果已部署的映像出現漏洞，則應盡快更換。

了解具有漏洞的映像部署位置對於保護環境安全至關重要。雖然您可以自行建置映像追蹤解決方案，但已有數個商業產品提供此功能和其他進階功能，包括：

- Grype
- Palo Alto - Prisma Cloud (twistcli)
- Aqua
- Kubei
- Trivy
- Snyk

Kubernetes 驗證 Webhook 也可以用來驗證映像沒有重大漏洞。驗證 Webhook 會在 Kubernetes API 之前叫用。它們通常用於拒絕不符合 Webhook 中定義的驗證條件的請求。這是無伺服器 Webhook 的

範例，可呼叫 ECR describeImageScanFindings API 來判斷 Pod 是否正在提取具有關鍵漏洞的映像。如果找到漏洞，則會拒絕 Pod，並傳回包含 CVEs 清單的訊息做為事件。

使用證明來驗證成品完整性

證明是以密碼編譯方式簽署的「陳述式」，宣告某些項目 -「述詞」，例如 管道執行或 SBOM 或漏洞掃描報告，對於另一個物件為 true，即「主體」，即 容器映像。

聲明可協助使用者驗證成品是否來自軟體供應鏈中的信任來源。例如，我們可能會使用容器映像，而不知道該映像中包含的所有軟體元件或相依性。不過，如果我們信任容器映像的生產者所說的軟體，我們可以使用生產者的證明來依賴該成品。這表示我們可以繼續在工作流程中安全地使用成品，而不是自行進行分析。

- 您可以使用 AWS Signer 或 Sigstore cosign 建立證明。
- Kyverno 等 Kubernetes 許可控制器可用來驗證證明。
- 請參閱本研討會，進一步了解如何使用開放原始碼工具在 AWS 上使用軟體供應鏈管理最佳實務，主題包括建立和連接證明至容器映像。

建立 ECR 儲存庫的 IAM 政策

現在，組織在共用 AWS 帳戶中擁有多個獨立運作的開發團隊並不常見。如果這些團隊不需要共用資產，您可能想要建立一組 IAM 政策，限制存取每個團隊可以互動的儲存庫。實作此項目的好方法是使用 ECR 命名空間。命名空間是將類似的儲存庫分組在一起的一種方式。例如，團隊 A 的所有登錄檔都可以在 team-a/ 前面，而團隊 B 的登錄檔可以使用 team-b/ 字首。限制存取的政策可能如下所示：

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowPushPull",
      "Effect": "Allow",
      "Action": [
        "ecr:GetDownloadUrlForLayer",
        "ecr:BatchGetImage",
        "ecr:BatchCheckLayerAvailability",
        "ecr:PutImage",
        "ecr:InitiateLayerUpload",
        "ecr:UploadLayerPart",
        "ecr:CompleteLayerUpload"
      ],
```

```
      "Resource": [
        "arn:aws:ecr:us-east-1:123456789012:repository/team-a/*"
      ]
    }
  ]
}
```

考慮使用 ECR 私有端點

ECR API 具有公有端點。因此，只要請求已經過 IAM 驗證和授權，就可以從網際網路存取 ECR 登錄檔。對於需要在叢集 VPC 缺少網際網路閘道 (IGW) 的沙盒環境中操作的人員，您可以設定 ECR 的私有端點。建立私有端點可讓您透過私有 IP 地址私下存取 ECR API，而不是透過網際網路路由流量。如需本主題的詳細資訊，請參閱 Amazon ECR 介面 VPC 端點。

實作 ECR 的端點政策

的預設端點政策允許存取區域內的所有 ECR 儲存庫。這可能會允許攻擊者/內部人員將資料封裝為容器映像，並將其推送到另一個 AWS 帳戶中的登錄檔，藉此滲透資料。降低此風險需要建立端點政策，以限制對 ECR 儲存庫的 API 存取。例如，下列政策允許帳戶中的所有 AWS 原則針對您的 執行所有動作，並且只針對您的 ECR 儲存庫執行所有動作：

```
{
  "Statement": [
    {
      "Sid": "LimitECRAccess",
      "Principal": "*",
      "Action": "*",
      "Effect": "Allow",
      "Resource": "arn:aws:ecr:<region>:<account_id>:repository/*"
    }
  ]
}
```

您可以透過設定使用新`PrincipalOrgID`屬性的條件來進一步增強這一點，以防止透過不屬於您 AWS Organization 的 IAM 原則推送/提取映像。如需其他詳細資訊，請參閱 aws：PrincipalOrgID。建議對 `com.amazonaws.<region>.ecr.dkr` 和 `com.amazonaws.<region>.ecr.api` 端點套用相同的政策。由於 EKS 從 ECR 提取 kube-proxy、Coredns 和 aws-node 的映像，因此您需要將登錄檔的帳戶 ID 新增至端點政策中的資源清單， `602401143452.dkr.ecr.us-west-2.amazonaws.com/` 或變更政策以允許提取並限制推送至您的帳戶 ID。下表顯示 EKS 映像來源的 AWS 帳戶與叢集區域之間的映射。

帳戶號碼	區域
602401143452	所有商業區域，但下列區域除外
—	—
800184023465	ap-east-1 - 亞太區域（香港）
558608220178	me-south-1 - 中東（巴林）
918309763551	cn-north-1 - 中國（北京）
961992271922	cn-northwest-1 - 中國（寧夏）

如需使用端點政策的詳細資訊，請參閱使用 VPC 端點政策來控制 Amazon ECR 存取。

實作 ECR 的生命週期政策

NIST 應用程式容器安全指南會警告「登錄檔中的過時映像」的風險，請注意，隨著時間的推移，應該移除具有易受攻擊、out-of-date軟體套件的舊映像，以防止意外部署和暴露。每個 ECR 儲存庫都可以有一個生命週期政策，設定映像過期時的規則。AWS 官方文件說明如何設定測試規則、進行評估，然後套用它們。官方文件中有數個生命週期政策範例，顯示篩選儲存庫中映像的不同方式：

- 依影像存留期或計數篩選
- 依已標記或未標記的映像進行篩選
- 在多個規則或單一規則中，依影像標籤進行篩選

？？？+ 警告 如果從 ECR 清除長時間執行應用程式的映像，當應用程式重新部署或水平擴展時，可能會導致映像提取錯誤。使用映像生命週期政策時，請確定您擁有良好的 CI/CD 實務，以保持部署及其參考的映像為最新狀態，並一律建立 【映像】 過期規則，以說明您執行發行/部署的頻率。

建立一組精選映像

與其允許開發人員建立自己的映像，請考慮為組織中的不同應用程式堆疊建立一組經過審核的映像。透過執行此動作，開發人員可以放棄學習如何編寫 Dockerfile 並專注於編寫程式碼。當變更合併到主節點時，CI/CD 管道可以自動編譯資產，將其存放在成品儲存庫中，並將成品複製到適當的映像，然後再推送到 Docker 登錄檔，例如 ECR。您至少應該建立一組基礎映像，讓開發人員從中建立自己的

Dockerfile。理想情況下，您想要避免從 Dockerhub 提取映像，因為 1/ 您不一定知道映像中的內容，而 2/ 大約五分之一的前 1000 個映像有漏洞。您可以在此處找到這些映像及其漏洞的清單。

將 USER 指令新增至您的 Dockerfiles，以非根使用者身分執行

如 Pod 安全區段中所述，您應該避免將容器做為根執行。雖然您可以將此設定為 podSpec 的一部分，但最好使用 Dockerfiles 的USER指令。USER 指令會將 UID 設定為在執行 USER 指令之後出現的 ENTRYPOINT、 RUN或 CMD指令時使用。

填入您的 Dockerfiles

內嵌可用來驗證您的 Dockerfiles 是否遵循一組預先定義的準則，例如 包含 USER指令或所有映像都必須加上標籤的要求。 dockerfile_lint 是來自 RedHat 的開放原始碼專案，可驗證常見的最佳實務，並包含規則引擎，您可以用來建置自己的 Dockerfiles 規則。它可以整合到 CI 管道中，其中使用違反規則的 Dockerfile 建置會自動失敗。

從 Scratch 建置映像

減少容器映像的攻擊面，應該是建置映像時的主要目標。執行此作業的理想方法是建立不含二進位檔的最小映像，以利用漏洞。幸運的是，Docker 具有從 建立映像的機制scratch。使用 Go 等語言，您可以建立靜態連結二進位檔，並在 Dockerfile 中參考它，如本範例所示：

```
############################
# STEP 1 build executable binary
############################
FROM golang:alpine AS builder# Install git.
# Git is required for fetching the dependencies.
RUN apk update && apk add --no-cache gitWORKDIR $GOPATH/src/mypackage/myapp/COPY . . #
 Fetch dependencies.
# Using go get.
RUN go get -d -v# Build the binary.
RUN go build -o /go/bin/hello

############################
# STEP 2 build a small image
############################
FROM scratch# Copy our static executable.
COPY --from=builder /go/bin/hello /go/bin/hello# Run the hello binary.
ENTRYPOINT ["/go/bin/hello"]
```

這會建立包含您的應用程式和任何其他項目的容器映像，使其非常安全。

搭配 ECR 使用不可變標籤

不可變標籤會強制您在每次推送至映像儲存庫時更新映像標籤。這可以阻止攻擊者以惡意版本覆寫映像，而不會變更映像的標籤。此外，它可讓您輕鬆且唯一地識別影像。

簽署您的映像、SBOMs、管道執行和漏洞報告

第一次推出 Docker 時，沒有用於驗證容器映像的密碼編譯模型。使用 v2 時，Docker 會將摘要新增至映像資訊清單。這允許對影像的組態進行雜湊，並使雜湊用於產生影像的 ID。啟用映像簽署時，Docker 引擎會驗證資訊清單的簽章，確保內容是從信任的來源產生，而且不會發生竄改。下載每個 layer 之後，引擎會驗證 layer 的摘要，確保內容符合資訊清單中指定的內容。影像簽署可讓您透過驗證與影像相關聯的數位簽章，有效地建立安全的供應鏈。

我們可以使用 AWS Signer 或 Sigstore Cosign 來簽署容器映像、建立 SBOMs的證明、漏洞掃描報告和管道執行報告。這些證明可確保映像的可信度和完整性，事實上它是由受信任管道建立的，沒有任何干擾或篡改，而且它只包含由映像發佈者驗證和信任的文件化軟體元件（在 SBOM 中）。這些證明可以連接到容器映像並推送到儲存庫。

在下一節中，我們將了解如何使用已證明的成品進行稽核和許可控制器驗證。

使用 Kubernetes 許可控制器進行映像完整性驗證

在使用動態許可控制器將映像部署至目標 Kubernetes 叢集之前，我們可以自動驗證映像簽章、已證明的成品，並只在成品的安全中繼資料符合許可控制器政策時，才允許部署。

例如，我們可以撰寫政策，以密碼編譯方式驗證映像的簽章、已驗證的 SBOM、已驗證的管道執行報告或已驗證的 CVE 掃描報告。我們可以在政策中寫入條件來檢查報告中的資料，例如 CVE 掃描不應有任何重要的 CVEs。只有滿足這些條件的映像才允許部署，所有其他部署都會遭到許可控制器拒絕。

許可控制器的範例包括：

- 基佛諾
- OPA Gatekeeper
- Portieris
- Ratify
- Kritis
- Grafeas 教學課程

- 抵用券

更新容器映像中的套件

您應該在 Dockerfiles `apt-get update && apt-get upgrade`中包含 RUN，以升級映像中的套件。雖然升級需要您以根身分執行，但在映像建置階段期間會發生這種情況。應用程式不需要以根執行。您可以安裝更新，然後使用 USER 指令切換到不同的使用者。如果您的基礎映像以非根使用者身分執行，請切換到根和後；不要只依賴基礎映像的維護器來安裝最新的安全性更新。

執行 從 `apt-get clean`刪除安裝程式檔案`/var/cache/apt/archives/`。您也可以在安裝套件`rm -rf /var/lib/apt/lists/*`後執行 。這會移除索引檔案或可供安裝的套件清單。請注意，每個套件管理員的這些命令可能不同。例如：

```
RUN apt-get update && apt-get install -y \
    curl \
    git \
    libsqlite3-dev \
    && apt-get clean && rm -rf /var/lib/apt/lists/*
```

工具和資源

- Amazon EKS 安全浸入研討會 - 映像安全
- docker-slim Build 安全最小映像
- dockle 驗證您的 Dockerfile 是否符合建立安全映像的最佳實務
- Dockerfiles 的 dockerfile-lint 規則型 linter
- hadolint 智慧型 dockerfile linter
- Gatekeeper 和 OPA 以政策為基礎的許可控制器
- Kyverno Kubernetes 原生政策引擎
- 傳入 允許使用者驗證供應鏈中的步驟是否打算執行，以及步驟是否由正確的執行者執行
- 公證人 用於簽署容器映像的專案
- 公證人 v2
- Grafeas 開放成品中繼資料 API，用於稽核和管理軟體供應鏈
- NeuVector by SUSE 開放原始碼、零信任容器安全平台，提供容器、映像和登錄檔掃描，以找出漏洞、秘密和合規。

多帳戶策略

AWS 建議使用多帳戶策略和 AWS 組織來協助隔離和管理業務應用程式和資料。使用多帳戶策略有許多好處：

- 增加 AWS API 服務配額。配額會套用至 AWS 帳戶，而針對工作負載使用多個帳戶會增加工作負載可用的整體配額。
- 簡化 Identity and Access Management (IAM) 政策。授予工作負載和運算子僅支援他們存取自己的 AWS 帳戶，意味著減少制定精細 IAM 政策以實現最低權限原則的時間。
- 改善 AWS 資源的隔離。根據設計，帳戶內佈建的所有資源都會邏輯上與其他帳戶中佈建的資源隔離。此隔離界限可讓您限制應用程式相關問題、組態錯誤或惡意動作的風險。如果一個帳戶內發生問題，可以減少或消除對其他帳戶中所含工作負載的影響。
- 更多好處，如 AWS 多帳戶策略白皮書所述

下列各節將說明如何使用集中式或非集中式 EKS 叢集方法，為您的 EKS 工作負載實作多帳戶策略。

規劃多租戶叢集的多工作負載帳戶策略

在多帳戶 AWS 策略中，屬於指定工作負載的資源，例如 S3 儲存貯體、ElastiCache 叢集和 DynamoDB Tables，都會在包含該工作負載所有資源的 AWS 帳戶中建立。這些稱為工作負載帳戶，EKS 叢集會部署到稱為叢集帳戶的帳戶中。叢集帳戶將在下一節中探索。將資源部署到專用工作負載帳戶類似於將 kubernetes 資源部署到專用命名空間。

工作負載帳戶接著可以依軟體開發生命週期或其他適當需求進一步細分。例如，指定的工作負載可以擁有生產帳戶、開發帳戶，或在特定區域中託管該工作負載執行個體的帳戶。如需詳細資訊，請參閱本 AWS 白皮書。

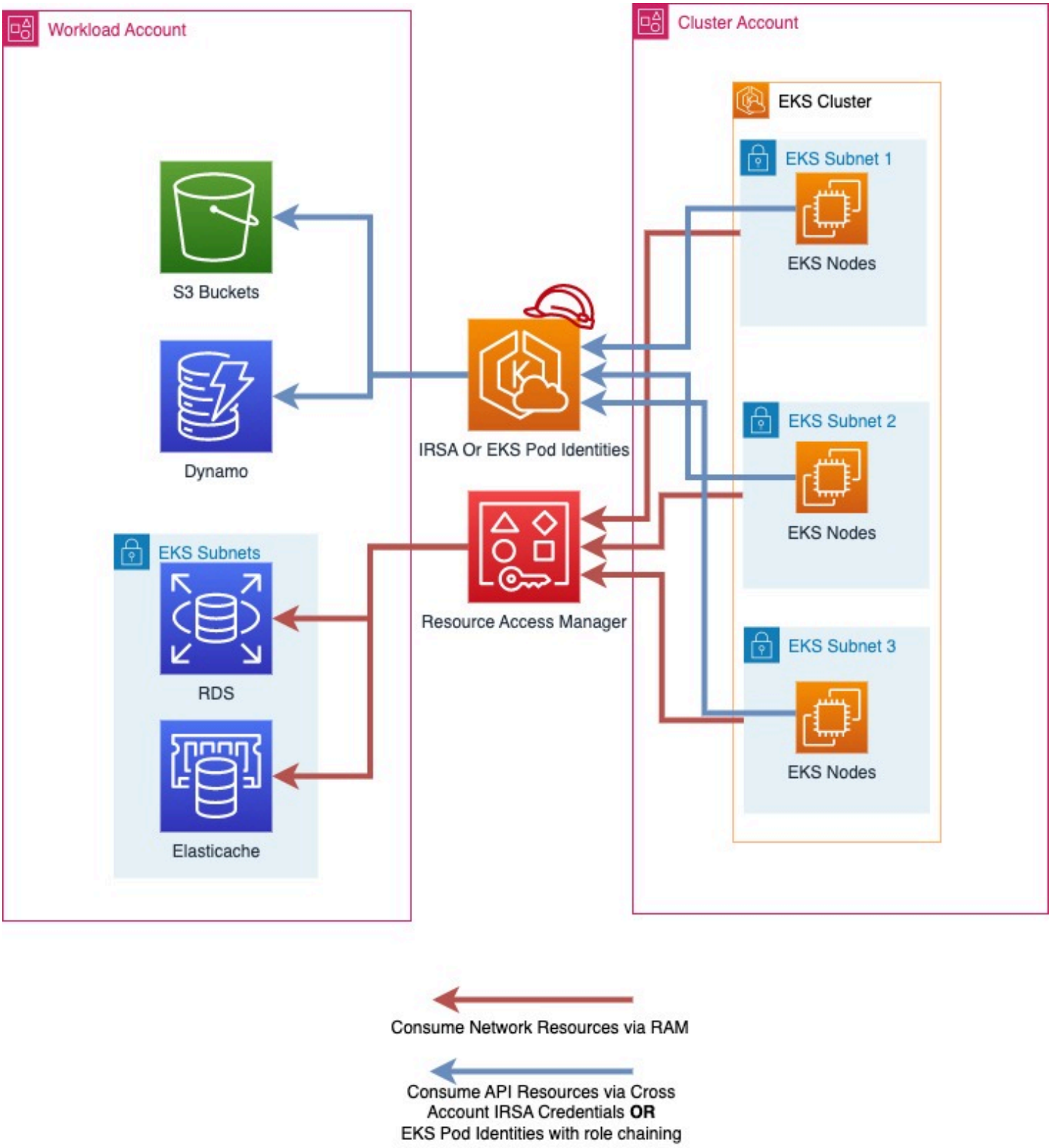
您可以在實作 EKS 多帳戶策略時採用下列方法：

集中式 EKS 叢集

在此方法中，您的 EKS 叢集將部署在名為 的單一 AWS 帳戶中`Cluster Account`。使用服務帳戶 (IRSA) 或 EKS Pod 身分的 IAM 角色交付臨時 AWS 登入資料和 AWS Resource Access Manager (RAM) 來簡化網路存取，您可以為多租戶 EKS 叢集採用多帳戶策略。 https://docs.aws.amazon.com/eks/latest/userguide/pod-identities.html叢集帳戶將包含 VPC、子網路、EKS 叢集、EC2/Fargate 運算資源 （ 工作者節點 ），以及執行 EKS 叢集所需的任何其他聯網組態。

在多租戶叢集的多工作負載帳戶策略中，AWS 帳戶通常會與 kubernetes 命名空間保持一致，做為隔離資源群組的機制。為多租戶 EKS 叢集實作多帳戶策略時，仍應遵循 EKS 叢集內的租戶隔離最佳實務。

您可以在 Cluster Accounts AWS 組織中擁有多個 ，而且最佳實務是擁有多個 Cluster Accounts，以符合您的軟體開發生命週期需求。對於大規模操作的工作負載，您可能需要多個 Cluster Accounts ，以確保有足夠的 kubernet 和 AWS 服務配額可供所有工作負載使用。


Workload Account
Cluster Account
EKS Cluster
EKS Subnet 1
EKS Nodes
S3 Buckets
Dynamo
IRSA Or EKS Pod Identities
EKS Subnet 2
EKS Nodes
EKS Subnets
Resource Access Manager
RDS
EKS Subnet 3
EKS Nodes
Elasticache
Consume Network Resources via RAM
Consume API Resources via Cross Account IRSA Credentials OR EKS Pod Identities with role chaining

|在上圖中，AWS RAM 用於將子網路從叢集帳戶共用到工作負載帳戶。然後，在 EKS Pod 中執行的工作負載會使用 IRSA 或 EKS Pod 身分和角色鏈結，在其工作負載帳戶中擔任角色並存取其 AWS 資源。

為多租戶叢集實作多工作負載帳戶策略

與 AWS Resource Access Manager 共用子網路

AWS Resource Access Manager (RAM) 可讓您跨 AWS 帳戶共用資源。

如果您的 AWS Organization 已啟用 RAM，您可以將 VPC 子網路從叢集帳戶共用至工作負載帳戶。這將允許工作負載帳戶擁有的 AWS 資源，例如 Amazon ElastiCache 叢集或 Amazon Relational Database Service (RDS) 資料庫，部署到與 EKS 叢集相同的 VPC 中，並可供在 EKS 叢集上執行的工作負載使用。

若要透過 RAM 共用資源，請在叢集帳戶的 AWS 主控台中開啟 RAM，然後選取「資源共用」和「建立資源共用」。為您的資源共用命名，然後選取您要共用的子網路。再次選取下一步，然後輸入您要與其共用子網路之工作負載帳戶的 IDs，再選取一次，然後按一下建立資源共用以完成。在此步驟之後，工作負載帳戶可以將資源部署到這些子網路。

RAM 共用也可以以程式設計方式建立，或使用基礎設施做為程式碼。

在 EKS Pod 身分和 IRSA 之間進行選擇

在 re：Invent 2023 上，AWS 啟動了 EKS Pod 身分，作為在 EKS 上將臨時 AWS 登入資料交付至 Pod 的更簡單方式。IRSA 和 EKS Pod 身分都是將臨時 AWS 登入資料交付至 EKS Pod 的有效方法，將持續受到支援。您應該考慮哪種交付方法最符合您的需求。

使用 EKS 叢集和多個 AWS 帳戶時，IRSA 可以直接在 EKS 叢集託管帳戶以外的 AWS 帳戶中擔任角色，而 EKS Pod 身分需要您設定角色鏈結。如需深入比較，請參閱 EKS 文件。

使用服務帳戶的 IAM 角色存取 AWS API 資源

服務帳戶的 IAM 角色 (IRSA) 可讓您將臨時 AWS 登入資料交付至在 EKS 上執行的工作負載。IRSA 可用來從叢集帳戶取得工作負載帳戶中 IAM 角色的臨時登入資料。這可讓您在叢集帳戶中的 EKS 叢集上執行的工作負載，以無預警的方式取用 AWS API 資源，例如工作負載帳戶中託管的 S3 儲存貯體，並針對 Amazon RDS 資料庫或 Amazon EFS FileSystems 等資源使用 IAM 身分驗證。

在工作負載帳戶中使用 IAM 身分驗證的 AWS API 資源和其他資源只能由相同工作負載帳戶中 IAM 角色的憑證存取，除非跨帳戶存取能夠且已明確啟用。

啟用 IRSA 進行跨帳戶存取

若要讓叢集帳戶中工作負載的 IRSA 存取工作負載帳戶中的資源，您必須先在工作負載帳戶中建立 IAM OIDC 身分提供者。這可以透過設定 IRSA 的相同程序來完成，但將在工作負載帳戶中建立身分提供者。

然後，在 EKS 上設定工作負載的 IRSA 時，您可以遵循與文件相同的步驟，但使用工作負載帳戶的 12 位數帳戶 ID，如「從另一個帳戶的叢集建立身分提供者範例」一節所述。

設定完成後，在 EKS 中執行的應用程式將能夠直接使用其服務帳戶來擔任工作負載帳戶中的角色，並使用其中的資源。

使用 EKS Pod 身分存取 AWS API 資源

EKS Pod 身分提供另一種方式，可將臨時 AWS 登入資料交付至在 EKS 上執行的工作負載。EKS Pod 身分與 EKS 控制平面和叢集上代理程式整合，讓 Pod 接收登入資料，而不需要您建立或管理 IAM OIDC 身分提供者。對於支援節點類型上的新工作負載，EKS Pod 身分是建議的方法，而 IRSA 仍然是完全支援的替代方案。

使用 EKS Pod 身分，您可以將叢集中的 Kubernetes 服務帳戶與叢集所在相同 AWS 帳戶中的 IAM 角色建立關聯。EKS 使用此關聯來取得該 IAM 角色的臨時登入資料，並將其安全地交付給使用該服務帳戶的 Pod。然後，您的應用程式可以使用標準 AWS SDKs和預設登入資料鏈來呼叫 AWS APIs；不需要自訂登入資料提供者或組態檔案。

啟用跨帳戶存取的 EKS Pod 身分

EKS Pod 身分透過在工作負載帳戶和 IAM 角色鏈結中使用目標 IAM 角色，原生支援跨帳戶存取。當您為 Kubernetes 服務帳戶建立 Pod Identity 關聯時，您可以在叢集帳戶中指定 Pod IAM 角色，並在工作負載帳戶中指定目標 IAM 角色。EKS Pod Identity 使用 Pod 角色擔任目標角色，並將目標角色的臨時登入資料傳回 Pod。

如需此方法的詳細演練和考量事項，請參閱此 AWS 部落格。

跨帳戶存取的 ABAC 和 EKS Pod 身分

使用 EKS Pod 身分作為多帳戶策略的一部分在其他帳戶中擔任角色（角色鏈結）時，您可以選擇為每個需要存取另一個帳戶的服務帳戶指派唯一的 IAM 角色，或跨多個服務帳戶使用通用 IAM 角色，並使用 ABAC 控制其可存取的帳戶。

若要使用 ABAC 來控制哪些服務帳戶可以使用角色鏈結來擔任另一個帳戶的角色，您可以建立角色信任政策陳述式，當預期值存在時，僅允許來自 EKS 叢集帳戶（帳戶 ID 111122223333) 的 Pod

IAM 角色擔任角色。下列角色信任政策只會讓來自 EKS 叢集帳戶的 Pod IAM 角色在 kubernetes-service-accounteks-cluster-arn和 kubernetes-namespace標籤都具有預期值時擔任角色。

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::111122223333:role/account-a-role"
            },
            "Action": [
                "sts:AssumeRole",
                "sts:TagSession"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:RequestTag/kubernetes-service-account": "PayrollApplication",
                    "aws:RequestTag/eks-cluster-arn": "arn:aws:eks:us-
east-1:111122223333:cluster/ProductionCluster",
                    "aws:RequestTag/kubernetes-namespace": "PayrollNamespace"
                }
            }
        }
    ]
}
```

您也可以使用 EKS Pod Identity 工作階段政策進一步縮小 Pod 的許可範圍，而無需建立其他 IAM 角色。當 EKS Pod Identity 擔任角色且產生的許可是角色政策和工作階段政策的交集時，會套用工作階段政策；如需考量和詳細步驟，請參閱 Amazon EKS Pod Identity 的 AWS Containers 部落格工作階段政策。

使用此策略時，最佳實務是確保常見的 IAM 角色只有 sts:AssumeRole和 sts:TagSession許可，且沒有其他 AWS 存取。

使用 ABAC 時，請務必控制誰能夠將 IAM 角色、使用者和 STS 工作階段標記為僅需要標記的人員。具有設定 kubernetes- 或 eks- 標籤功能的人，可以設定與 EKS Pod Identity 角色鏈結期間傳遞的標籤相同的標籤，也可以提升其權限。您可以限制誰有權使用 IAM 政策或服務控制政策 (SCP) 設定這些標籤；例如控制項，請參閱 ProtectPodIdentitiesTagsOnRolesAndUsers 和 STS-Protect-EKS-pod-identities-tags。

在 EKS Pod 身分和 IRSA 之間進行選擇

IRSA 和 EKS Pod 身分都是將臨時 AWS 登入資料交付至 EKS 工作負載的有效選項。對於在支援的節點類型上執行的新應用程式，建議使用 EKS Pod 身分，而且 IRSA 非常適合您已有 OIDC 和 IRSA，或在 EKS Pod 身分不支援的平台上執行。

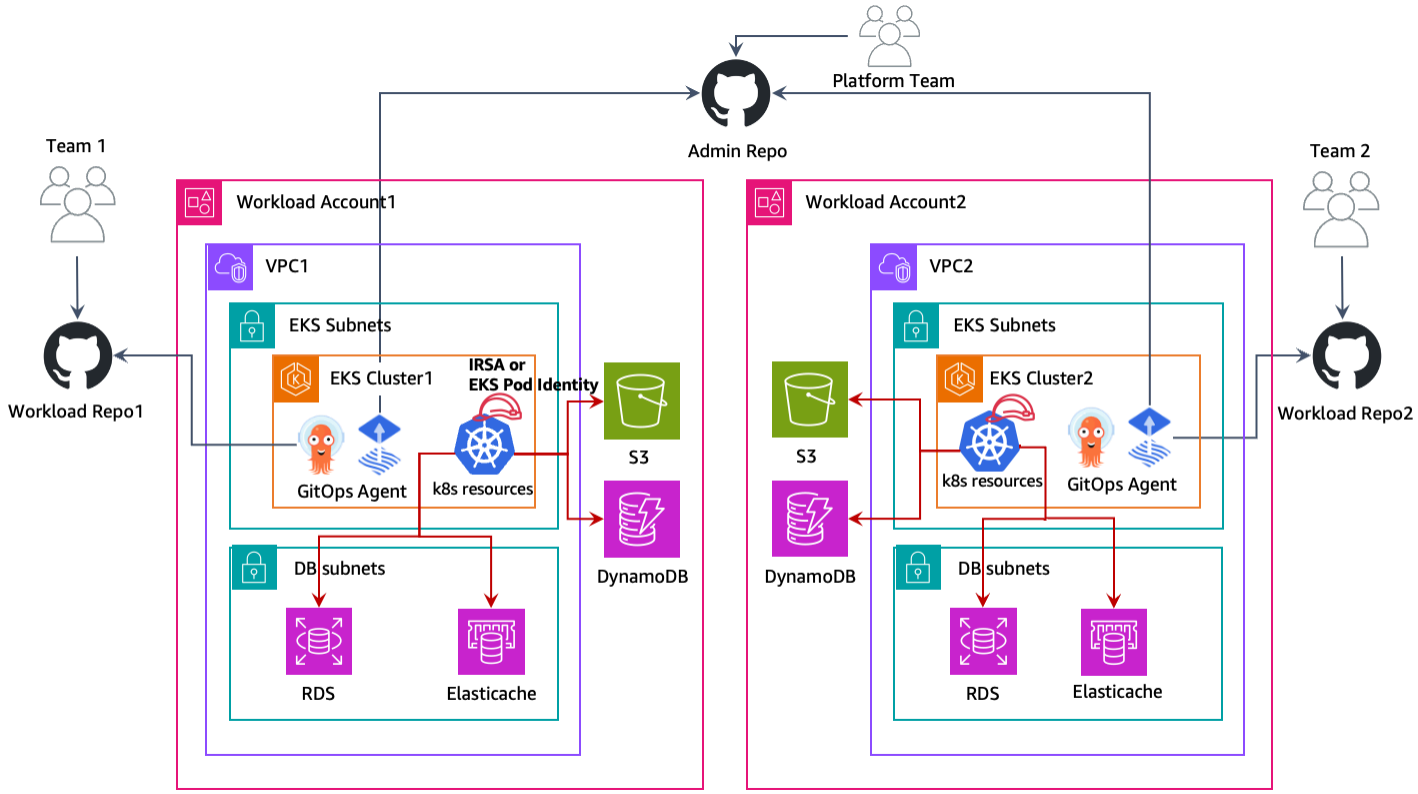
在決定要使用哪個時，請考慮下列事項：

- 在下列情況下選擇 EKS Pod 身分：
  - 您正在設計新的工作負載，並希望避免建立和管理 IAM OIDC 身分提供者。
  - 您想要使用目標 IAM 角色進行跨帳戶存取的原生支援，而無需將自訂登入資料指令碼或 AWS 組態檔案新增至您的 Pod。
  - 您偏好叢集管理員管理哪些 Kubernetes 服務帳戶可以擔任哪些角色，而 IAM 管理員則管理這些角色的許可。
  - 您希望憑證販賣有效率地擴展，並避免達到 IAM Quota 限制。
- 在下列情況下選擇 IRSA：
  - 您已成功使用 IRSA，並具有 OIDC 提供者和 IAM 角色的標準模式。
  - 您的工作負載會在不支援 EKS Pod 身分的環境中執行，例如 AWS Fargate、Windows 節點，或使用不支援的 AWS SDKs的應用程式。
  - 您需要工作負載帳戶中角色的直接 OIDC 型聯合模型，而且您的安全控制已經以 OIDC 供應商為基礎建置。

IRSA 和 EKS Pod 身分都支援多帳戶策略。您可以在叢集中一致地使用方法，或採用混合模型，其中舊版工作負載會繼續使用 IRSA，而新工作負載則會使用 EKS Pod 身分。

分散式 EKS 叢集

在此方法中，EKS 叢集會部署到各自的工作負載 AWS 帳戶，並與其他 AWS 資源一起運作，例如 Amazon S3 儲存貯體、VPCs、Amazon DynamoDB 資料表等，每個工作負載帳戶都是獨立的、自給自足的，並由各自的業務單位/應用程式團隊運作。此模型允許為各種叢集功能建立可重複使用的藍圖 ： AI/ML 叢集、批次處理、一般用途等， 並根據應用程式團隊需求提供叢集。應用程式和平台團隊都會在各自的 GitOps 儲存庫之外操作，以管理工作負載叢集的部署。



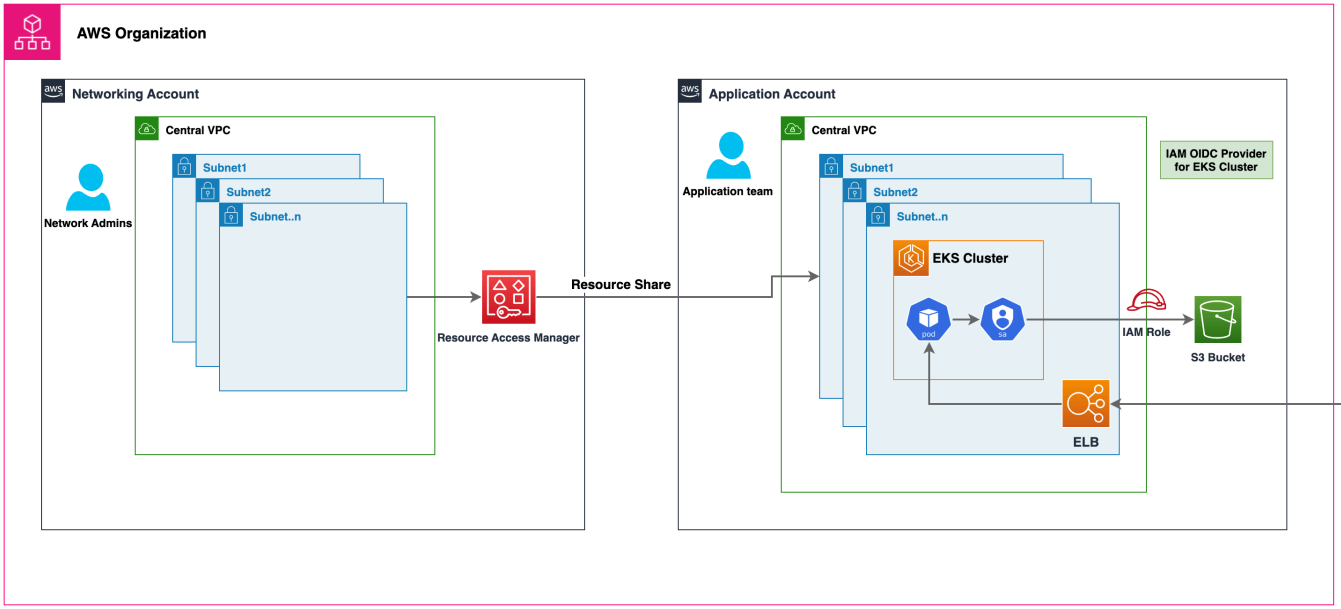
在上圖中，Amazon EKS 叢集和其他 AWS 資源會部署到各自的工作負載帳戶。然後，在 EKS Pod 中執行的工作負載會使用 IRSA 或 EKS Pod 身分來存取其 AWS 資源。

GitOps 是管理應用程式和基礎設施部署的一種方式，以便在 Git 儲存庫中宣告地描述整個系統。這是一種操作模型，可讓您使用版本控制、不可變成品和自動化的最佳實務來管理多個 Kubernetes 叢集的狀態。在此多叢集模型中，每個工作負載叢集都會使用多個 Git 儲存庫啟動，允許每個團隊（應用程式、平台、安全性等）在叢集上部署各自的變更。

您可以在每個帳戶中使用服務帳戶 (IRSA) 或 EKS Pod 身分的 IAM 角色，讓您的 EKS 工作負載取得臨時 aws 登入資料，以安全地存取其他 AWS 資源。 https://docs.aws.amazon.com/eks/latest/userguide/pod-identities.html在個別工作負載 AWS 帳戶中建立 IAM 角色，並將其映射至 k8s 服務帳戶，以提供暫時 IAM 存取。因此，此方法不需要跨帳戶存取。遵循服務帳戶的 IAM 角色文件，了解如何在 IRSA 的每個工作負載中設定 ，以及 EKS Pod 身分文件，了解如何在每個帳戶中設定 EKS Pod 身分。

集中式聯網

您也可以利用 AWS RAM 將 VPC 子網路共用至工作負載帳戶，並在其中啟動 Amazon EKS 叢集和其他 AWS 資源。這可實現集中式網路管理/管理、簡化的網路連線，以及分散式 EKS 叢集。如需此方法的詳細演練和考量，請參閱此 AWS 部落格。



在上圖中，AWS RAM 用於將子網路從中央聯網帳戶共用到工作負載帳戶。然後，EKS 叢集和其他 AWS 資源會在個別工作負載帳戶中的這些子網路中啟動。EKS Pod 使用 IRSA 或 EKS Pod 身分來存取其 AWS 資源。

集中式與分散式 EKS 叢集

使用集中式或非集中式執行 的決定將取決於您的需求。此表示範每個策略的主要差異。

#	集中式 EKS 叢集	分散式 EKS 叢集
叢集管理：	管理單一 EKS 叢集比管理多個叢集更容易	為了降低管理多個 EKS 叢集的操作負荷，需要有效率的叢集管理自動化
成本效益：	允許重複使用 EKS 叢集和網路資源，以提高成本效益	需要每個工作負載的聯網和叢集設定，這需要額外的資源
彈性：	如果叢集受損，集中式叢集上的多個工作負載可能會受到影響	如果叢集受損，則損壞僅限於在該叢集上執行的工作負載。所有其他工作負載不受影響
隔離與安全：	隔離/軟多租用戶是使用 k8s 原生建構達成的，例如 Namespaces 。工作負載	當工作負載在未共用任何資源的個別叢集和節點中執行時，對運算資源進行更強大的隔

#	集中式 EKS 叢集	分散式 EKS 叢集
	可能會共用基礎資源，例如 CPU、記憶體等。AWS 資源會隔離到自己的工作負載帳戶，預設情況下無法從其他 AWS 帳戶存取。	離。AWS 資源會隔離到自己的工作負載帳戶，預設情況下無法從其他 AWS 帳戶存取。
效能與擴展：	隨著工作負載擴展到非常大的規模，您可能會在叢集帳戶中遇到 kubernet 和 AWS 服務配額。您可以部署額外的叢集帳戶以進一步擴展	隨著存在更多叢集和 VPCs，每個工作負載都有更多可用的 k8s 和 AWS 服務配額
網路：	每個叢集使用單一 VPC，為該叢集上的應用程式提供更簡單的連線	必須在分散式 EKS 叢集 VPCs 之間建立路由
Kubernetes Access Management：	需要在叢集中維護許多不同的角色和使用者，以便為所有工作負載團隊提供存取權，並確保已正確隔離 kubernetes 資源	簡化存取管理，因為每個叢集專用於工作負載/團隊
AWS Access Management：	AWS 資源會部署到自己的帳戶，預設只能透過工作負載帳戶中的 IAM 角色存取。工作負載帳戶中的 IAM 角色會擔任具有 IRSA 或 EKS Pod 身分的跨帳戶。	AWS 資源會部署到自己的帳戶，預設只能透過工作負載帳戶中的 IAM 角色存取。工作負載帳戶中的 IAM 角色會直接交付至具有 IRSA 或 EKS Pod 身分的 Pod

叢集存取管理

有效的存取管理對於維護 Amazon EKS 叢集的安全性和完整性至關重要。本指南探索 EKS 存取管理的各種選項，重點是使用 AWS IAM Identity Center（先前稱為 AWS SSO)。我們將比較不同的方法、討論其權衡，並強調已知的限制和考量。

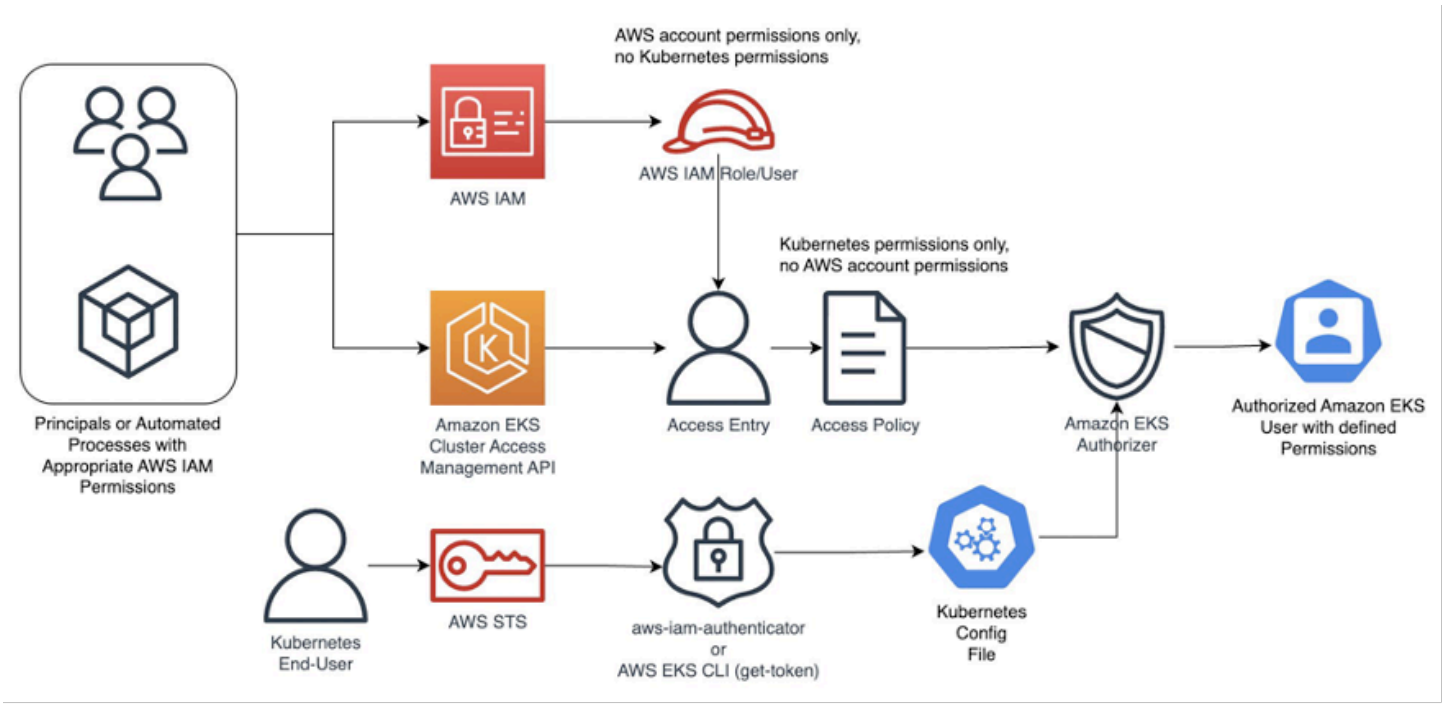
EKS 存取管理選項

> **Note**
>
> ConfigMap 型存取管理 (aws-auth ConfigMap) 已被叢集存取管理 (CAM) API 取代。對於新的 EKS 叢集，實作 CAM API 來管理叢集存取。對於使用 aws-auth ConfigMap 的現有叢集，請使用 CAM API 遷移至 。

選項 1：具有叢集存取管理 (CAM) API 的 AWS IAM Identity Center

- 集中式使用者和許可管理
- 與現有身分提供者整合（例如 Microsoft AD、Okta、PingId 等）
- CAM API 使用存取項目將 AWS IAM 主體（使用者或角色）連結至 EKS 叢集。這些項目適用於 IAM Identity Center 的受管身分，可讓管理員控制 Identity Center 中定義之使用者和群組的叢集存取。

EKS 叢集身分驗證流程：



1. 委託人（人類使用者）或自動化程序透過 AWS IAM 透過提供適當的 AWS 帳戶許可進行身分驗證。在此步驟中，它們會映射到適當的 AWS IAM 主體（角色或使用者）。

2. 接著，EKS 存取項目會定義適當的存取政策，僅包含 Kubernetes 許可，將此 IAM 主體映射至 Kubernetes RBAC 主體（使用者或群組）。
3. 當 Kubernetes 最終使用者嘗試存取叢集時，其身分驗證請求會由 aws-iam-authenticator 或 AWS EKS CLI 處理，並根據 kubeconfig 檔案中的叢集內容進行驗證。
4. 最後，EKS 授權方會驗證與已驗證使用者存取項目相關聯的許可，並相應地授予或拒絕存取。
   - API 使用 Amazon EKS 特定的存取政策來定義每個存取項目的授權層級。這些政策可以映射到 IAM Identity Center 中設定的角色和許可，確保跨 AWS 服務和 EKS 叢集的一致存取控制。

ConfigMap 型存取管理的優勢：

1. 降低組態錯誤的風險：直接 API 型管理可消除與手動 ConfigMap 編輯相關的常見錯誤。這有助於防止意外刪除或語法錯誤，可能會將使用者封鎖在叢集之外。
2. 增強型最低權限原則：從叢集建立者身分中移除叢集管理員許可的需求，並允許更精細且適當的許可指派。您可以選擇為打破玻璃的使用案例新增此許可。
3. 增強型安全模型：在套用存取項目之前，提供存取項目的內建驗證。此外， 提供更緊密的 AWS IAM 整合以進行身分驗證。
4. 簡化操作：提供更直覺的方式，透過 AWS 原生工具管理許可。

最佳實務：

1. 使用 AWS Organizations 管理多個帳戶並套用服務控制政策 SCPs)。
2. 為不同的 EKS 角色（例如，管理員、開發人員、唯讀） 建立特定許可集，以實作最低權限原則。
3. 利用屬性型存取控制 (ABAC)，根據使用者屬性將許可動態指派給 Pod。
4. 定期稽核和檢閱存取許可。

考量/限制：

1. Identity Center 產生的角色 ARNs 具有隨機尾碼，使其在靜態組態中使用具有挑戰性。
2. 在 Kubernetes 資源層級對精細許可的有限支援。自訂 Kubernetes RBAC 角色需要額外的組態。除了 Kubernetes 原生 RBAC 之外，請考慮在 EKS 叢集中使用 Kyverno 進行進階許可管理。

選項 2：映射至 Kubernetes 群組的 AWS IAM 使用者/角色

優點：

1. 精細控制 IAM 許可。
2. 可預測和靜態角色 ARNs

缺點：

1. 使用者帳戶的管理開銷增加
2. 缺乏集中式身分管理
3. IAM 實體擴散的可能性

最佳實務：

1. 使用 IAM 角色而非 IAM 使用者以提高安全性和管理性
2. 為角色實作命名慣例，以確保一致性和易於管理
3. 利用 IAM 政策條件，根據標籤或其他屬性限制存取。
4. 定期輪換存取金鑰並檢閱許可。

考量/限制：

1. 管理大量使用者或角色時的可擴展性問題
2. 沒有內建的單一登入功能

選項 3：OIDC 供應商

優點：

1. 與現有身分管理系統整合
2. 降低使用者帳戶的管理開銷

缺點：

1. 其他組態複雜性
2. 身分驗證期間延遲增加的可能性
3. 外部身分提供者的相依性

最佳實務：

1. 仔細設定 OIDC 提供者，以確保安全字符驗證。
2. 使用短期字符並實作字符重新整理機制。
3. 定期稽核和更新 OIDC 組態。

檢閱本指南，了解整合外部單一登入供應商與 Amazon EKS 的參考實作

考量/限制：

1. 相較於 IAM，與 AWS 服務的原生整合有限。
2. OIDC 供應商的發行者 URL 必須可公開存取，EKS 才能探索簽署金鑰。

工作負載的 AWS EKS Pod 身分與 IRSA

Amazon EKS 提供兩種方式將 AWS IAM 許可授予在 Amazon EKS 叢集中執行的工作負載：服務帳戶的 IAM 角色 (IRSA) 和 EKS Pod 身分。

雖然 IRSA 和 EKS Pod 身分都提供最低權限存取、憑證隔離和可稽核性的優點，但 EKS Pod Identity 是將許可授予工作負載的建議方法。

如需 EKS Pod 身分和登入資料的詳細指引，請參閱安全最佳實務中的身分和登入資料一節。

建議

結合 IAM Identity Center 與 CAM API

- 簡化管理：透過使用叢集存取管理 API 搭配 IAM Identity Center，管理員可以與其他 AWS 服務一起管理 EKS 叢集存取，減少在不同介面之間切換或手動編輯 ConfigMaps 的需求。

- 使用存取項目，從叢集外部管理 IAM 主體的 Kubernetes 許可。您可以使用 EKS API、AWS 命令列界面、AWS SDKs來新增和管理對叢集的存取。 AWS CloudFormation 這意味著您可以使用建立叢集時使用的工具來管理使用者。
- 精細 Kubernetes 許可可以透過存取項目和存取政策，與與 SSO 身分相關聯的 IAM 主體映射 Kubernetes 使用者或群組套用。
- 若要開始使用，請依照變更身分驗證模式來使用存取項目，然後遷移現有的 aws-auth ConfigMap 項目來存取項目。

叢集自動擴展的最佳實務

本指南提供有關 Cluster Autoscaling 的建議，包括 Auto Mode、Karpenter 和 Kubernetes Cluster Autoscaler 的指引。

主題

- EKS 自動模式
- Karpenter
- Cluster Autoscaler

EKS 自動模式

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

Amazon EKS Auto Mode 代表 Kubernetes 基礎設施管理的重大發展，結合安全且可擴展的叢集基礎設施與 AWS 管理的整合式 Kubernetes 功能。此服務提供全受管的工作者節點操作，無需客戶設定受管節點群組或 AutoScaling 群組。

關鍵的架構差異在於 EKS Auto Mode 使用 Karpenter 型系統，自動佈建 EC2 執行個體以回應 Pod 請求 。這些執行個體會在具有預先安裝附加元件的 Bottlerocket AMIs 上執行，例如 EBS CSI 驅動程式，讓基礎設施真正由 AWS 管理。與傳統擴展方法相反：

- 傳統 Cluster Autoscaler (CAS) 需要手動節點群組管理，而且每個節點群組只能建立具有單一執行個體類型的節點
- 自我管理的 Karpenter 透過使用 EC2 機群 API 提供更多彈性，並且可以佈建不同的執行個體類型，但需要客戶管理
- EKS Auto Mode 會透過受管 NodePools 和 NodeClasses 自動處理所有擴展操作

新系統推出數個操作改進：

- 無需手動節點群組組態的自動 Pod 驅動擴展
- 內建的受管負載平衡器控制器，可根據輸入資源自動建立 ALB/NLB

- 具有預先設定 Pod 身分的整合式安全功能
- 節點執行時間上限 21 天，含自動取代

從成本角度來看，EKS Auto Mode 會維持標準 EC2 定價，同時僅針對 Auto Mode 受管節點新增管理費。重要的是，客戶仍然可以將 Auto Mode 受管節點與相同叢集 中的自我管理節點混合。

雖然 AWS 處理大多數操作層面，但客戶仍需負責叢集版本管理，並可執行可觸發工作者節點滾動更新的受控升級。

使用自動模式的原因

自動模式是針對想要 Kubernetes 和 EKS 優點，但需要將 Kubernetes 的操作負擔降至最低的使用者，例如自動擴展、負載平衡和儲存等關鍵平台組件的升級和安裝/維護。自動模式進一步將 EKS 與 Kubernetes 維護一起進行的未區分繁重任務最小化

常見問答集

EKS Auto Mode 和開放原始碼 Karpenter 之間的差異是什麼？

EKS Auto Mode 是一套大型功能，可讓執行生產級 Kubernetes 變得簡單。其中一個功能是全受管 Karpenter 的自動擴展優勢。從操作角度來看，唯一的差異在於 EKS Auto Mode，您不需要管理 Karpenter Pod 本身的部署、擴展和升級。受管 NodeClasses 和 NodePools 等所有其他操作的運作方式與開放原始碼 Karpenter 的運作方式相同。

我可以與 Auto Mode 受管節點一起執行受管節點群組嗎？

是，您可以透過受管節點群組與 Auto Mode 提供的自動擴展節點一起執行靜態節點

我可以將叢集從標準 EKS 遷移到 EKS 自動模式嗎？

是，可在官方 AWS 文件中找到在現有叢集上啟用 EKS Auto Mode 的說明

注意事項：1. 啟用自動模式後，您會想要解除安裝已安裝且現在由自動模式管理的任何元件，例如 Karpenter 或 AWS Load Balancer 控制器 2。您需要確保已安裝的附加元件是up-to-date。請參閱 文件。

如何在 EKS Auto 模式中設定 NodePools？

新叢集將預先設定兩個 NodePools

一般用途

```yaml
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  generation: 1
  labels:
    app.kubernetes.io/managed-by: eks
  name: general-purpose
spec:
  disruption:
    budgets:
      - nodes: 10%
    consolidateAfter: 30s
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    spec:
      expireAfter: 336h
      nodeClassRef:
        group: eks.amazonaws.com
        kind: NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values:
            - on-demand
        - key: eks.amazonaws.com/instance-category
          operator: In
          values:
            - c
            - m
            - r
        - key: eks.amazonaws.com/instance-generation
          operator: Gt
          values:
            - '4'
        - key: kubernetes.io/arch
          operator: In
          values:
            - amd64
        - key: kubernetes.io/os
          operator: In
          values:
            - linux
      terminationGracePeriod: 24h0m0s
```

此 NodePool 會指示 Karpenter 啟動具有下列特性的節點：

1. 「隨需」的容量類型
2. C、M 或 R 的執行個體類型
3. 執行個體產生 4
4. AMD 架構
5. Linux 作業系統

它還透過宣告只有 10% 的所有節點在任何指定時間都可能處於中斷狀態，並且只有在節點是空的或未充分利用時，才會發生合併，來定義縮減邏輯是什麼。

system

```yaml
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  generation: 1
  labels:
    app.kubernetes.io/managed-by: eks
  name: system
spec:
  disruption:
    budgets:
      - nodes: 10%
    consolidateAfter: 30s
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    spec:
      expireAfter: 336h
      nodeClassRef:
        group: eks.amazonaws.com
        kind: NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values:
            - on-demand
        - key: eks.amazonaws.com/instance-category
          operator: In
          values:
            - c
            - m
            - r
        - key: eks.amazonaws.com/instance-generation
          operator: Gt
          values:
            - '4'
        - key: kubernetes.io/arch
          operator: In
          values:
            - amd64
            - arm64
        - key: kubernetes.io/os
          operator: In
          values:
            - linux
      taints:
        - effect: NoSchedule
```

此 NodePool 類似於「一般用途」，但下列差異除外：

1. 它允許具有 ARM 架構和 AMD 架構的節點
2. 除非對「CriticalAddonsOnly」有公差，否則它會使用 NoSchedule 對這些節點進行污點。這是供 EKS 附加元件內部使用

自訂

您可以根據您的需求建立自己的自訂 NodePools。若要進一步了解 NodePools，請參閱 Karpenter 文件。

啟動新節點時，我可以自訂自動模式使用的 AMI 嗎？

否，目前唯一支援的 AMIs 適用於 Amazon 提供的 Bottlerocket

如何在 Kubernetes 主機上安裝自訂工具或代理程式？

由於不支援 AMI 自訂，如果您需要主機層級軟體進行安全性掃描等操作，您應該將工作負載部署為 Kubernetes DaemonSet。

當我佈建新的 EKS Auto Mode 叢集時，哪些元件正在我的叢集資料平面中執行？

如果您使用 eksctl 或 AWS 主控台建立叢集，EKS Auto Mode 叢集中執行的唯一 Pod 是 Kubernetes Metrics Server Pod。EKS Auto Mode 的其他元件，例如 Karpenter、AWS Load Balancer 控制器和 EBS CSI 驅動程式，都在叢集外執行和管理。

正在執行哪些受管元件以支援我的新 EKS Auto Mode 叢集？

EKS Auto Mode 可完全自動化部署生產級 Kubernetes 所需的大部分資料平面。其中包含：

- Karpenter，用於自動擴展叢集的運算
- AWS Load Balancer 控制器可讓您透過自動化 Elastic Load Balancer 整合輕鬆公開 Kubernetes 服務
- VPC CNI
- 叢集 DNS
- kube-proxy
- EBS CSI
- EKS Pod 身分識別代理程式

- EKS 節點監控代理程式

如何對用於在叢集中作為 Pod 執行的自動模式元件進行故障診斷？

使用 EKS Auto Mode 時，許多 AWS Load Balancer 控制器和 Karpenter 等元件都會在叢集之外為您管理，因此您無法從自我管理時使用的日誌中看到相同的可見性。如果您遇到需要疑難排解一段自動模式功能的功能的情況，請建立 AWS Support 票證。

Karpenter

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

Karpenter 是一種開放原始碼專案，旨在增強 Kubernetes 叢集內的節點生命週期管理。它會根據 Pod 的特定排程需求，自動佈建和取消佈建節點，從而實現高效的擴展和成本最佳化。其主要函數為：

- 監控 Kubernetes 排程器因為資源限制而無法排程的 Pod。
- 評估不可排程 Pod 的排程需求（資源請求、節點選擇器、親和性、公差等）。
- 佈建符合這些 Pod 需求的新節點。
- 不再需要節點時將其移除。

透過 Karpenter，您可以定義具有節點佈建限制的 NodePools，例如污點、標籤、需求（執行個體類型、區域等），以及佈建資源總數的限制。部署工作負載時，您可以在 Pod 規格中指定各種排程限制，例如資源請求/限制、節點選擇器、節點/Pod 親和性、容錯能力和拓撲分散限制。然後，Karpenter 將根據這些規格佈建適當大小的節點。

使用 Karpenter 的原因

在 Karpenter 推出之前，Kubernetes 使用者主要依賴 Amazon EC2 Auto Scaling 群組和 Kubernetes Cluster Autoscaler (CAS) 動態調整其叢集的運算容量。透過 Karpenter，您不需要建立數十個節點群組，即可透過 Karpenter 實現彈性和多樣性。與 CAS 不同，Karpenter 不會與 Kubernetes 版本緊密結合，也不需要您在 AWS 和 Kubernetes APIs 之間跳躍。

Karpenter 將執行個體協同運作責任合併在單一系統中，這更簡單、更穩定且能感知叢集。Karpenter 旨在透過提供簡化方法來克服 Cluster Autoscaler 提出的一些挑戰：

- 根據工作負載需求佈建節點。
- 使用靈活的 NodePool 選項，依執行個體類型建立多樣化的節點組態。Karpenter 可以讓您使用單一且靈活的 NodePool 管理各種工作負載容量，而不是管理許多特定的自訂節點群組。
- 透過快速啟動節點和排程 Pod，大規模實現改善的 Pod 排程。

如需有關使用 Karpenter 的資訊和文件，請造訪 karpenter.sh 網站。

建議

最佳實務分為 Karpenter 本身、NodePools 和 Pod 排程的區段。

Karpenter 最佳實務

下列最佳實務涵蓋與 Karpenter 本身相關的主題。

鎖定生產叢集中的 AMIs

我們強烈建議您鎖定 Karpenter 用於生產叢集的已知 Amazon Machine Image (AMIs)。使用 `amiSelector`搭配設為 的別名`@latest`，或使用一些其他方法，導致在發佈未測試AMIs 時部署，這可讓您的生產叢集有工作負載失敗和停機的風險。因此，我們強烈建議您在非生產叢集中測試較新版本的生產叢集時，鎖定生產叢集的 AMIs 測試工作版本。例如，您可以在 NodeClass 中設定別名，如下所示：

```
amiSelectorTerms
  - alias: al2023@v20240807
```

如需在 Karpenter 中管理和鎖定 AMIs 的資訊，請參閱 Karpenter 文件中的管理 AMIs。

針對容量需求不斷變化的工作負載使用 Karpenter

相較於 Autoscaling Groups (ASGs) 和 Managed Node Groups (MNGs)，Karpenter 讓擴展管理更接近 Kubernetes 原生 APIs。ASGs 和 MNGs 是 AWS 原生抽象，其中會根據 EC2 CPU 負載等 AWS 層級指標觸發擴展。Cluster Autoscaler 會將 Kubernetes 抽象化為 AWS 抽象，但會因此失去一些彈性，例如為特定可用區域排程。

Karpenter 移除了 AWS 抽象層，將一些靈活性直接帶入 Kubernetes。Karpenter 最適合用於工作負載遇到高峰值需求的叢集，或具有各種運算需求的叢集。MNGs 和 ASGs 適用於執行工作負載的叢集，這些工作負載往往更靜態且一致。您可以根據您的需求，混合使用動態和靜態受管節點。

考慮其他自動擴展專案，當...

您需要仍在 Karpenter 中開發的功能。由於 Karpenter 是相對較新的專案，如果您需要尚未屬於 Karpenter 的功能，請考慮目前其他自動擴展專案。

在 EKS Fargate 或屬於節點群組的工作者節點上執行 Karpenter 控制器

Karpenter 是使用 Helm Chart 安裝。Helm Chart 會將 Karpenter 控制器和 Webhook Pod 安裝為需要執行的部署，才能使用控制器擴展叢集。我們建議至少有一個小型節點群組，其中至少有一個工作者節點。或者，您可以建立 `karpenter` 命名空間的 Fargate 設定檔，在 EKS Fargate 上執行這些 Pod。這樣做會導致部署到此命名空間的所有 Pod 在 EKS Fargate 上執行。請勿在由 Karpenter 管理的節點上執行 Karpenter。

Karpenter 不支援自訂啟動範本

v1 APIs 沒有自訂啟動範本支援。您可以使用自訂使用者資料和/或直接在 EC2NodeClass 中指定自訂 AMIs。如需如何執行此操作的詳細資訊，請參閱 NodeClasses。

排除不符合工作負載的執行個體類型

如果叢集中執行的工作負載不需要特定執行個體類型，請考慮使用 `node.kubernetes.io/instance-type`金鑰將其排除。

下列範例示範如何避免佈建大型 Graviton 執行個體。

```
- key: node.kubernetes.io/instance-type
  operator: NotIn
  values:
  - m6g.16xlarge
  - m6gd.16xlarge
  - r6g.16xlarge
  - r6gd.16xlarge
  - c6g.16xlarge
```

使用 Spot 時啟用中斷處理

Karpenter 支援原生中斷處理，可以處理非自願中斷事件，例如 Spot 執行個體中斷、排定的維護事件、執行個體終止/停止可能中斷工作負載的事件。當 Karpenter 偵測到節點的此類事件時，它會自動事先污點、耗盡和終止受影響的節點，以在中斷之前開始正常清理工作負載。對於 2 分鐘通知的 Spot 中斷，Karpenter 會快速啟動新的節點，以便在回收執行個體之前移動 Pod。若要啟用中斷處理，您可

以使用為此目的佈建的 SQS 佇列名稱來設定 `--interruption-queue` CLI 引數。不建議搭配 Node Termination Handler 使用 Karpenter 中斷處理，如此處所述。

需要檢查點或其他形式正常耗盡的 Pod，在關機前需要 2 分鐘，應該會在其叢集中啟用 Karpenter 中斷處理。

沒有傳出網際網路存取的 Amazon EKS 私有叢集

將 EKS 叢集佈建至沒有網際網路路由的 VPC 時，您必須確定已根據 EKS 文件中出現的私有叢集需求來設定環境。此外，您需要確定已在 VPC 中建立 STS VPC 區域端點。如果沒有，您會看到類似以下顯示的錯誤。

```
{"level":"FATAL","time":"2024-02-29T14:28:34.392Z","logger":"controller","message":"Checking
 EC2 API connectivity, WebIdentityErr: failed to retrieve credentials
\ncaused by: RequestError: send request failed\ncaused by: Post
 \"https://sts.<region>.amazonaws.com/\": dial tcp 54.239.32.126:443: i/o
 timeout","commit":"596ea97"}
```

這些變更在私有叢集中是必要的，因為 Karpenter 控制器使用服務帳戶的 IAM 角色 (IRSA)。使用 IRSA 設定的 Pod 會透過呼叫 AWS Security Token Service (AWS STS) API 來取得登入資料。如果沒有傳出網際網路存取，您必須在 VPC 中建立並使用 AWS STS VPC 端點。

私有叢集也需要您為 SSM 建立 VPC 端點。當 Karpenter 嘗試佈建新節點時，它會查詢啟動範本組態和 SSM 參數。如果您的 VPC 中沒有 SSM VPC 端點，則會導致下列錯誤：

```
{"level":"ERROR","time":"2024-02-29T14:28:12.889Z","logger":"controller","message":"Unable
 to hydrate the AWS launch template cache, RequestCanceled: request context canceled
\ncaused by: context canceled","commit":"596ea97","tag-key":"karpenter.k8s.aws/
cluster","tag-value":"eks-workshop"}
...
{"level":"ERROR","time":"2024-02-29T15:08:58.869Z","logger":"controller.nodeclass","message":"d
 amis from ssm, getting ssm parameter \"/aws/service/eks/optimized-ami/1.27/amazon-
linux-2/recommended/image_id\", RequestError: send request failed\ncaused by:
 Post \"https://ssm.<region>.amazonaws.com/\": dial tcp 67.220.228.252:443: i/o
 timeout","commit":"596ea97","ec2nodeclass":"default","query":"/aws/service/eks/
optimized-ami/1.27/amazon-linux-2/recommended/image_id"}
```

價格清單查詢 API 沒有 VPC 端點。因此，定價資料會隨著時間而過時。Karpenter 透過在其二進位中包含隨需定價資料來解決這個問題，但只會在 Karpenter 升級時更新該資料。失敗的定價資料請求將導致以下錯誤訊息：

```
{"level":"ERROR","time":"2024-02-29T15:08:58.522Z","logger":"controller.pricing","message":"ret
 on-demand pricing data, RequestError: send request failed\ncaused by: Post
 \"https://api.pricing.<region>.amazonaws.com/\": dial tcp 18.196.224.8:443:
 i/o timeout; RequestError: send request failed\ncaused by: Post \"https://
api.pricing.<region>.amazonaws.com/\": dial tcp 18.185.143.117:443: i/o
 timeout","commit":"596ea97"}
```

請參閱本文件，在完全私有 EKS 叢集中使用 Karpenter，並了解要建立的 VPC 端點。

建立 NodePools

下列最佳實務涵蓋與建立 NodePools 相關的主題。

建立多個 NodePools 時...

當不同的團隊共用叢集，且需要在不同的工作者節點上執行工作負載，或具有不同的作業系統或執行個體類型需求時，請建立多個 NodePools 例如，一個團隊可能想要使用 Bottlerocket，而另一個團隊可能想要使用 Amazon Linux。同樣地，一個團隊可能可以存取另一個團隊不需要的昂貴 GPU 硬體。使用多個 NodePools 可確保每個團隊都能使用最適當的資產。

建立互斥或加權NodePools

建議建立互斥或加權的 NodePools，以提供一致的排程行為。如果它們不匹配且多個 NodePools 相符，Karpenter 會隨機選擇要使用的節點Pools，從而導致非預期的結果。建立多個 NodePools 的實用範例包括下列項目：

使用 GPU 建立 NodePool，且僅允許在這些（昂貴）節點上執行特殊工作負載：

```
# NodePool for GPU Instances with Taints
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu
spec:
  disruption:
    consolidateAfter: 1m
    consolidationPolicy: WhenEmptyOrUnderutilized
  template:
    metadata: {}
    spec:
      nodeClassRef:
```

```
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      expireAfter: Never
      requirements:
      - key: node.kubernetes.io/instance-type
        operator: In
        values:
        - p3.8xlarge
        - p3.16xlarge
      - key: kubernetes.io/os
        operator: In
        values:
        - linux
      - key: kubernetes.io/arch
        operator: In
        values:
        - amd64
      - key: karpenter.sh/capacity-type
        operator: In
        values:
        - on-demand
      taints:
      - effect: NoSchedule
        key: nvidia.com/gpu
        value: "true"
```

具有污點公差的部署：

```
# Deployment of GPU Workload will have tolerations defined
apiVersion: apps/v1
kind: Deployment
metadata:
  name: inflate-gpu
spec:
    spec:
      tolerations:
      - key: "nvidia.com/gpu"
        operator: "Exists"
        effect: "NoSchedule"
```

對於另一個團隊的一般部署，NodePool 規格可能包含 nodeAffinity。然後，部署可以使用 nodeSelectorTerms 來比對 `billing-team`。

```
# NodePool for regular EC2 instances
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: generalcompute
spec:
  template:
    metadata:
      labels:
        billing-team: my-team
    spec:
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      expireAfter: Never
      requirements:
      - key: node.kubernetes.io/instance-type
        operator: In
        values:
        - m5.large
        - m5.xlarge
        - m5.2xlarge
        - c5.large
        - c5.xlarge
        - c5a.large
        - c5a.xlarge
        - r5.large
        - r5.xlarge
      - key: kubernetes.io/os
        operator: In
        values:
        - linux
      - key: kubernetes.io/arch
        operator: In
        values:
        - amd64
      - key: karpenter.sh/capacity-type
        operator: In
        values:
        - on-demand
```

使用 nodeAffinity 部署：

```
# Deployment will have spec.affinity.nodeAffinity defined
kind: Deployment
metadata:
  name: workload-my-team
spec:
  replicas: 200
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: "billing-team"
                  operator: "In"
                  values: ["my-team"]
```

使用計時器 (TTL) 從叢集自動刪除節點

您可以在佈建節點上使用計時器來設定何時刪除沒有工作負載 Pod 或已達到過期時間的節點。節點過期可用作升級方法，以便淘汰節點並以更新版本取代。如需使用 `spec.template.spec` 設定節點過期的資訊，請參閱 Karpenter 文件中的過期。

避免過度限制 Karpenter 可以佈建的執行個體類型，尤其是在使用 Spot 時

使用 Spot 時，Karpenter 會使用 Price Capacity Optimized 配置策略來佈建 EC2 執行個體。此策略會指示 EC2 從最深的集區佈建執行個體，以取得您啟動的執行個體數量，並具有最低的中斷風險。EC2 機群接著會從這些集區的最低價格請求 Spot 執行個體。您允許 Karpenter 利用的執行個體類型越多，EC2 就越能最佳化 Spot 執行個體的執行時間。根據預設，Karpenter 將使用叢集部署所在區域和可用區域中的所有執行個體類型 EC2 優惠。Karpenter 會根據待定的 Pod 以智慧方式從所有執行個體類型中選擇，以確保您的 Pod 已排程在適當大小且配備的執行個體上。例如，如果您的 Pod 不需要 GPU，Karpenter 不會將您的 Pod 排程到支援 GPU 的 EC2 執行個體類型。當您不確定要使用的執行個體類型時，可以執行 Amazon ec2-instance-selector 來產生符合您運算需求的執行個體類型清單。例如，CLI 將記憶體 vCPU、架構和區域作為輸入參數，並為您提供滿足這些限制條件的 EC2 執行個體清單。

```
$ ec2-instance-selector --memory 4 --vcpus 2 --cpu-architecture x86_64 -r ap-
southeast-1
c5.large
c5a.large
c5ad.large
```

```
c5d.large
c6i.large
t2.medium
t3.medium
t3a.medium
```

使用 Spot 執行個體時，您不應該對 Karpenter 設定太多限制，因為這樣做可能會影響應用程式的可用性。例如，假設會回收特定類型的所有執行個體，而且沒有可用的替代選項可以取代它們。您的 Pod 將保持待定狀態，直到已設定執行個體類型的 Spot 容量被補充為止。您可以將執行個體分散到不同的可用區域，以減少容量不足錯誤的風險，因為 Spot 集區在 AZs 之間不同。也就是說，一般最佳實務是允許 Karpenter 在使用 Spot 時使用一組不同的執行個體類型。

排程 Pod

下列最佳實務與使用 Karpenter 在叢集中部署 Pod 進行節點佈建相關。

遵循 EKS 最佳實務以獲得高可用性

如果您需要執行高可用性應用程式，請遵循一般 EKS 最佳實務建議。如需如何跨節點和區域分散 Pod 的詳細資訊，請參閱 Karpenter 文件中的拓撲分散。使用中斷預算來設定需要維護的最小可用 Pod，以防嘗試移出或刪除 Pod。

使用分層限制條件來限制雲端供應商提供的運算功能

Karpenter 的分層限制模型可讓您建立一組複雜的 NodePool 和 Pod 部署限制，以取得 Pod 排程的最佳可能相符項目。Pod 規格可以請求的限制範例包括下列項目：

- 需要在只有特定應用程式可用的可用區域中執行。例如，假設您的 Pod 必須與位於特定可用區域的 EC2 執行個體上執行的另一個應用程式通訊。如果您的目標是減少 VPC 中的跨可用區流量，您可能想要在 EC2 執行個體所在的可用區中共同放置 Pod。這種類型的目標通常是使用節點選擇器來完成。如需節點選取器的詳細資訊，請參閱 Kubernetes 文件。
- 需要特定類型的處理器或其他硬體。如需需要在 GPU 上執行 Pod 的 Pod 規格範例，請參閱 Karpenter 文件的加速器一節。

建立帳單警示以監控您的運算支出

當您將叢集設定為自動擴展時，您應該建立帳單警示，在支出超過閾值時提醒您，並將資源限制新增至 Karpenter 組態。使用 Karpenter 設定資源限制類似於設定 AWS Autoscaling 群組的最大容量，因為它代表 Karpenter NodePool 可執行個體化的運算資源數量上限。

> **Note**
>
> 無法設定整個叢集的全域限制。限制適用於特定 NodePools。

以下程式碼片段會指示 Karpenter 佈建最多 1000 個 CPU 核心和 1000Gi 記憶體。只有在達到或超過限制時，Karpenter 才會停止新增容量。超過限制時，Karpenter 控制器會將 `memory resource usage of 1001 exceeds limit of 1000` 或類似的訊息寫入控制器的日誌。如果您要將容器日誌路由到 CloudWatch 日誌，您可以建立指標篩選條件來尋找日誌中的特定模式或詞彙，然後建立 CloudWatch 警示，以便在您設定的指標閾值違反時提醒您。

如需使用限制搭配 Karpenter 的詳細資訊，請參閱 Karpenter 文件中的設定資源限制。

```
spec:
  limits:
    cpu: 1000
    memory: 1000Gi
```

如果您未使用限制或限制 Karpenter 可以佈建的執行個體類型，Karpenter 會視需要繼續將運算容量新增至您的叢集。雖然以這種方式設定 Karpenter 可讓您的叢集自由擴展，但它也可能具有重大的成本影響。因此，我們建議您設定帳單警示。帳單警示可讓您在帳戶中計算的預估費用（超過定義的閾值）時收到提醒和主動通知。如需其他資訊，請參閱設定 Amazon CloudWatch 帳單警示以主動監控預估費用。

您可能也想要啟用成本異常偵測，這是一種 AWS 成本管理功能，使用機器學習來持續監控您的成本和用量，以偵測不尋常的支出。如需詳細資訊，請參閱 AWS 成本異常偵測入門指南。如果您已盡可能在 AWS Budgets 中建立預算，您也可以設定 動作，以便在違反特定閾值時通知您。透過預算動作，您可以傳送電子郵件、將訊息發佈至 SNS 主題，或將訊息傳送至聊天機器人，例如 Slack。如需詳細資訊，請參閱設定 AWS Budgets 動作。

使用 karpenter.sh/do-not-disrupt 註釋來防止 Karpenter 取消佈建節點

如果您在 Karpenter 佈建的節點上執行關鍵應用程式，例如長時間執行的批次任務或具狀態應用程式，且節點的 TTL 已過期，則該應用程式會在執行個體終止時中斷。透過將`karpenter.sh/do-not-disrupt`註釋新增至 Pod，您可以指示 Karpenter 保留節點，直到 Pod 終止或移除`karpenter.sh/do-not-disrupt`註釋為止。如需詳細資訊，請參閱中斷文件。

如果節點上保留的唯一非協助程式集 Pod 是與任務相關聯的 Pod，只要任務狀態成功或失敗，Karpenter 就能夠鎖定和終止這些節點。

設定請求 = 使用整合時所有非 CPU 資源的限制

透過比較 Pod 資源請求與節點上可配置資源的數量，進行一般性的合併和排程。不會考慮資源限制。例如，記憶體限制大於記憶體請求的 Pod 可能會爆量超過請求。如果同一節點上的多個 Pod 同時爆量，這可能會導致某些 Pod 由於記憶體不足 (OOM) 條件而終止。合併可以使這種情況更可能發生，因為它可以只考慮其請求將 Pod 封裝到節點上。

使用 LimitRanges 設定資源請求和限制的預設值

由於 Kubernetes 不會設定預設請求或限制，因此容器從基礎主機、CPU 和記憶體消耗的資源不受限制。Kubernetes 排程器會查看 Pod 的請求總數 (Pod 的容器請求總數或 Pod 的 Init 容器資源總數較高），以決定要排程 Pod 的工作者節點。同樣地，Karpenter 會考慮 Pod 的請求，以判斷其佈建的執行個體類型。您可以使用限制範圍來套用命名空間的合理預設值，以防某些 Pod 未指定資源請求。

請參閱設定命名空間的預設記憶體請求和限制

將準確的資源請求套用至所有工作負載

Karpenter 能夠在工作負載需求的相關資訊準確時，啟動最適合您工作負載的節點。如果使用 Karpenter 的合併功能，這尤其重要。

請參閱設定和調整所有工作負載的資源請求/限制大小

CoreDNS 建議

更新 CoreDNS 的組態以維護可靠性

在 Karpenter 管理的節點上部署 CoreDNS Pod 時，考慮到 Karpenter 在快速終止/建立新節點以符合需求的動態性質，建議遵循下列最佳實務：

CoreDNS lameduck 持續時間

CoreDNS 準備程度探查

這將確保 DNS 查詢不會導向尚未就緒或已終止的 CoreDNS Pod。

Karpenter 藍圖

由於 Karpenter 採取應用程式優先的方法來將 的運算容量佈建至 Kubernetes 資料平面，因此您可能想知道如何正確設定它們。Karpenter 藍圖是一種儲存庫，其中包含遵循此處所述最佳實務的常見工作負

載案例清單。您將擁有建立已設定 Karpenter 的 EKS 叢集所需的所有資源，並測試儲存庫中包含的每個藍圖。您可以結合不同的藍圖（最後建立工作負載所需的藍圖）。

其他資源

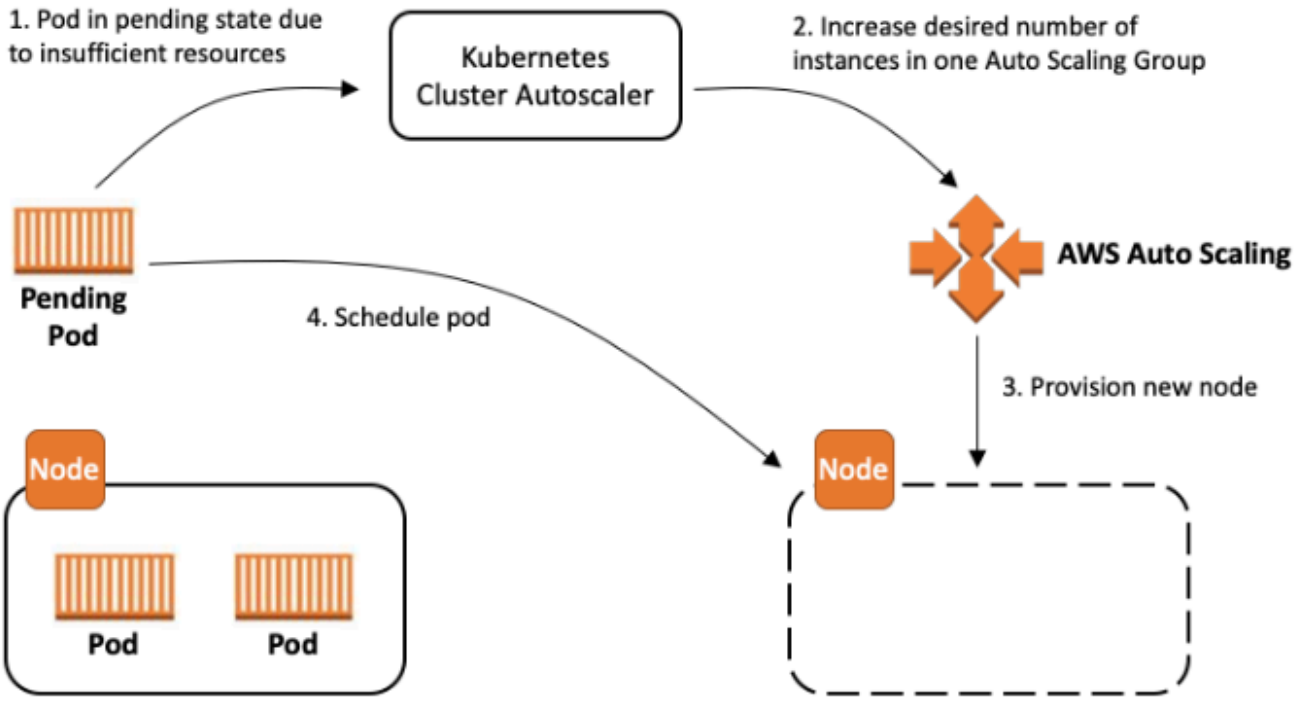
- Karpenter 沉浸式日研討會
- Karpenter 成本最佳化研討會
- EKS 研討會 - Karpenter
- Karpenter 與 Cluster Autoscaler
- 於 re：Invent 2023 的 Karpenter 工作階段
- 教學課程：使用 Amazon EC2 Spot 和 Karpenter 以更低的成本執行 Kubernetes 叢集

Cluster Autoscaler

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

概觀

Kubernetes Cluster Autoscaler 是由 SIG Autoscaling 維護的熱門 Cluster Autoscaling 解決方案。它負責確保您的叢集有足夠的節點來排程您的 Pod，而不會浪費資源。它會監控無法排程的 Pod，以及未充分利用的節點。然後，它會模擬在將變更套用至叢集之前新增或移除節點。Cluster Autoscaler 中的 AWS 雲端提供者實作會控制 EC2 Auto Scaling 群組`.DesiredReplicas`的欄位。



本指南將提供心理模型，用於設定 Cluster Autoscaler 並選擇符合組織需求的最佳權衡集。雖然沒有單一最佳組態，但有一組組態選項可讓您權衡效能、可擴展性、成本和可用性。此外，本指南將提供最佳化 AWS 組態的秘訣和最佳實務。

詞彙表

本文件中將經常使用以下術語。這些術語可以有廣泛的意義，但僅限於以下定義，以用於本文件的目的。

可擴展性是指 Cluster Autoscaler 在 Kubernetes Cluster 數量增加時的效能。達到可擴展性限制時，Cluster Autoscaler 的效能和功能會降低。當 Cluster Autoscaler 超過其可擴展性限制時，可能不會再新增或移除叢集中的節點。

效能是指 Cluster Autoscaler 能夠做出和執行擴展決策的速度。完美執行的 Cluster Autoscaler 會立即做出決策並觸發擴展動作以回應刺激，例如 Pod 變得無法排程。

可用性表示 Pod 可以快速排程，而不會中斷。這包括當新建立的 Pod 需要排程時，以及縮減節點終止為其排程的任何剩餘 Pod 時。

成本取決於橫向擴展和縮減事件背後的決策。如果現有節點未充分利用，或新增了對傳入 Pod 太大的新節點，則會浪費資源。根據使用案例，由於積極的縮減規模決策，提早終止 Pod 可能會產生相關成本。

節點群組是叢集內一組節點的抽象 Kubernetes 概念。它不是真正的 Kubernetes 資源，但以抽象形式存在於 Cluster Autoscaler、叢集 API 和其他元件中。節點群組內的節點會共用標籤和污點等屬性，但可能包含多個可用區域或執行個體類型。

EC2 Auto Scaling 群組可用於 EC2 上的節點群組實作。EC2 Auto Scaling 群組設定為啟動執行個體，以自動加入其 Kubernetes 叢集，並將標籤和污點套用到 Kubernetes API 中對應的節點資源。

EC2 受管節點群組是 EC2 上節點群組的另一個實作。它們可消除手動設定 EC2 Autoscaling 擴展群組的複雜性，並提供節點版本升級和正常節點終止等其他管理功能。

操作 Cluster Autoscaler

Cluster Autoscaler 通常會安裝為您叢集中的部署。它使用領導者選擇來確保高可用性，但工作一次由單一複本完成。它無法水平擴展。對於基本設定，預設應該使用提供的安裝說明開箱即用，但有幾件事需要記住。

請確認以下事項：

- Cluster Autoscaler 的版本符合叢集的版本。未測試或支援跨版本相容性。
- 除非您有阻止使用此模式的特定進階使用案例，否則會啟用自動探索。

使用最低權限存取 IAM 角色

使用 Auto Discovery 時，強烈建議您透過限制動作`autoscaling:SetDesiredCapacity`和範圍為目前叢集的 Auto Scaling 群組`autoscaling:TerminateInstanceInAutoScalingGroup`來使用最低權限存取。

這將防止在一個叢集中執行的 Cluster Autoscaler 修改不同叢集中的節點群組，即使`--node-group-auto-discovery`引數沒有使用標籤縮小到叢集的節點群組（例如 `k8s.io/cluster-autoscaler/<cluster-name>`)。

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "autoscaling:SetDesiredCapacity",
                "autoscaling:TerminateInstanceInAutoScalingGroup"
            ],
            "Resource": "*",
```

```
            "Condition": {
                "StringEquals": {
                    "aws:ResourceTag/k8s.io/cluster-autoscaler/enabled": "true",
                    "aws:ResourceTag/k8s.io/cluster-autoscaler/my-cluster": "owned"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "autoscaling:DescribeAutoScalingGroups",
                "autoscaling:DescribeAutoScalingInstances",
                "autoscaling:DescribeLaunchConfigurations",
                "autoscaling:DescribeScalingActivities",
                "autoscaling:DescribeTags",
                "ec2:DescribeImages",
                "ec2:DescribeInstanceTypes",
                "ec2:DescribeLaunchTemplateVersions",
                "ec2:GetInstanceTypesFromInstanceRequirements",
                "eks:DescribeNodegroup"
            ],
            "Resource": "*"
        }
    ]
}
```

設定您的節點群組

有效的自動擴展從為您的叢集正確設定一組節點群組開始。選取正確的節點群組集是將工作負載的可用性最大化並降低成本的關鍵。AWS 使用 EC2 Auto Scaling 群組實作節點群組，這些群組可靈活處理大量使用案例。不過，Cluster Autoscaler 會對您的節點群組做出一些假設。讓您的 EC2 Auto Scaling 群組組態與這些假設保持一致，可將不需要的行為降至最低。

請確認以下事項：

- 節點群組中的每個節點都有相同的排程屬性，例如標籤、標記和資源。
  - 對於 MixedInstancePolicies，CPU、記憶體和 GPU 的執行個體類型必須具有相同的形狀
  - 政策中指定的第一個執行個體類型將用於模擬排程。
  - 如果您的政策有其他具有更多資源的執行個體類型，則縮減後可能會浪費資源。
  - 如果您的政策具有資源較少的其他執行個體類型，則 Pod 可能無法在執行個體上排程。
- 具有多個節點的節點群組優先於具有較少節點的多個節點群組。這將對可擴展性產生最大的影響。

- 盡可能在兩個系統提供支援時偏好 EC2 功能 （ 例如 區域、MixedInstancePolicy)

> **Note**
>
> 建議使用 EKS 受管節點群組。受管節點群組隨附強大的管理功能，包括 Cluster Autoscaler 的功能，例如自動 EC2 Auto Scaling 群組探索和正常節點終止。

針對效能和可擴展性進行最佳化

了解自動擴展演算法的執行時間複雜性，將協助您調整 Cluster Autoscaler，以便在具有超過 1，000 個節點的大型叢集中繼續順暢操作。

用於調校 Cluster Autoscaler 可擴展性的主要旋鈕是提供給程序的資源、演算法的掃描間隔，以及叢集中的節點群組數量。此演算法的實際執行時間複雜性涉及其他因素，例如排程外掛程式複雜性和 Pod 數量。這些被視為無法設定的參數，因為這些參數對叢集的工作負載而言是自然的，而且無法輕鬆調校。

Cluster Autoscaler 會將整個叢集的狀態載入記憶體，包括 Pod、節點和節點群組。在每個掃描間隔，演算法會識別無法排程的 Pod，並模擬每個節點群組的排程。調校這些因素有不同的權衡，應仔細考慮您的使用案例。

垂直自動擴展叢集自動擴展器

將 Cluster Autoscaler 擴展至大型叢集的最簡單方法是增加其部署的資源請求。大型叢集的記憶體和 CPU 都應該增加，但這會因叢集大小而有很大差異。自動擴展演算法會將所有 Pod 和節點存放在記憶體中，在某些情況下，這可能會導致記憶體佔用空間大於 1 GB。增加資源通常手動完成。如果您發現持續的資源調校正在造成操作負擔，請考慮使用附加元件恢復器或垂直 Pod Autoscaler。

減少節點群組的數量

將節點群組數量降至最低，是確保 Cluster Autoscaler 能在大型叢集上繼續正常運作的一種方式。對於每個團隊或每個應用程式建構其節點群組的一些組織而言，這可能具有挑戰性。雖然 Kubernetes API 完全支援此功能，但這被視為具有可擴展性影響的 Cluster Autoscaler 反模式。使用多個節點群組 （ 例如 Spot 或 GPUs) 有許多原因，但在許多情況下，有替代設計可在使用少量群組時達到相同的效果。

請確認以下事項：

- Pod 隔離是使用命名空間而非節點群組來完成。

  - 這可能無法在低信任的多租戶叢集中實現。
  - Pod ResourceRequests 和 ResourceLimits 已正確設定，以避免資源爭用。
  - 較大的執行個體類型將產生更理想的儲存貯體封裝，並減少系統 Pod 額外負荷。
- NodeTaints 或 NodeSelectors 用於將 Pod 排程為例外狀況，而非規則。
- 區域資源定義為具有多個可用區域的單一 EC2 Auto Scaling 群組。

減少掃描間隔

低掃描間隔（例如 10 秒）可確保 Cluster Autoscaler 在 Pod 變得無法排程時盡快回應。不過，每次掃描都會對 Kubernetes API 和 EC2 Auto Scaling 群組或 EKS 受管節點群組 APIs 產生許多 API 呼叫。這些 API 呼叫可能會導致 Kubernetes 控制平面的速率限制，甚至無法使用服務。

預設掃描間隔為 10 秒，但在 AWS 上，啟動節點需要更長的時間才能啟動新的執行個體。這表示可以增加間隔，而不會顯著增加整體擴充規模時間。例如，如果啟動節點需要 2 分鐘，將間隔變更為 1 分鐘將導致 6 倍的 API 呼叫減少，以降低 38% 的擴展速度。

跨節點群組碎片

Cluster Autoscaler 可以設定為在特定的一組節點群組上操作。使用此功能，您可以部署 Cluster Autoscaler 的多個執行個體，每個執行個體都設定為在不同的一組節點群組上操作。此策略可讓您任意使用大量的節點群組，以交易成本實現可擴展性。我們只建議使用此做為改善效能的最後手段。

Cluster Autoscaler 最初不是為此組態設計，因此有一些副作用。由於碎片無法通訊，因此多個自動擴展器可能會嘗試排程無法排程的 Pod。這可能會導致不必要地向外擴展多個節點群組。這些額外的節點會在 之後縮減`scale-down-delay`。

```
metadata:
  name: cluster-autoscaler
  namespace: cluster-autoscaler-1

...

--nodes=1:10:k8s-worker-asg-1
--nodes=1:10:k8s-worker-asg-2

---

metadata:
  name: cluster-autoscaler
  namespace: cluster-autoscaler-2
```

```
...

--nodes=1:10:k8s-worker-asg-3
--nodes=1:10:k8s-worker-asg-4
```

請確認以下事項：

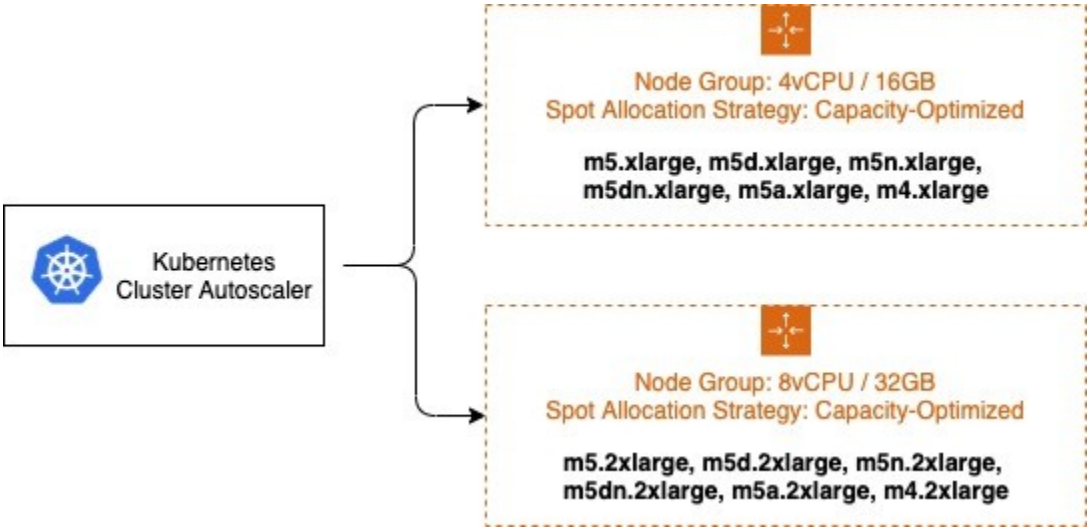
- 每個碎片都設定為指向一組唯一的 EC2 Auto Scaling 群組
- 每個碎片都會部署到個別的命名空間，以避免領導者選擇衝突

最佳化成本和可用性

Spot 執行個體

您可以在節點群組中使用 Spot 執行個體，並節省高達隨需價格 90% 的折扣，而且當 EC2 需要恢復容量時，Spot 執行個體可以隨時中斷權衡。當您的 EC2 Auto Scaling 群組因為缺乏可用容量而無法擴展時，會發生容量不足錯誤。透過選取多個執行個體系列來最大化多樣性，可以利用許多 Spot 容量集區來增加實現所需擴展的機會，並減少 Spot 執行個體中斷對叢集可用性的影響。具有 Spot 執行個體的混合執行個體政策是在不增加節點群組數量的情況下增加多樣性的絕佳方式。請記住，如果您需要保證的資源，請使用隨需執行個體而非 Spot 執行個體。

設定混合執行個體政策時，所有執行個體類型都必須具有類似的資源容量。自動擴展器的排程模擬器使用 MixedInstancePolicy 中的第一個 InstanceType。如果後續執行個體類型較大，則擴展後可能會浪費資源。如果較小，由於容量不足，您的 Pod 可能無法在新執行個體上排程。例如，M4、M5、M5a 和 M5n 執行個體都有類似的 CPU 和記憶體數量，並且是 MixedInstancePolicy 的理想候選項目。EC2 Instance Selector 工具可協助您識別類似的執行個體類型。



建議將隨需和 Spot 容量隔離為不同的 EC2 Auto Scaling 群組。這優於使用基本容量策略，因為排程屬性基本上不同。由於 Spot 執行個體隨時中斷（當 EC2 需要恢復容量時），使用者通常會污點其可先佔節點，因此需要對先佔行為的明確 Pod 容錯能力。這些污點會導致節點的排程屬性不同，因此它們應該分成多個 EC2 Auto Scaling 群組。

Cluster Autoscaler 具有 Expanders 的概念，提供不同的策略來選擇要擴展的節點群組。策略`--expander=least-waste`是良好的一般用途預設值，如果您要使用多個節點群組進行 Spot 執行個體多樣化（如上圖所述），它可以透過擴展在擴展活動後最適合使用的群組來進一步最佳化節點群組的成本。

排定節點群組 / ASG 的優先順序

您也可以使用優先順序擴展器來設定優先順序型自動擴展。 `--expander=priority`可讓您的叢集排定節點群組/ASG 的優先順序，如果因為任何原因而無法擴展，則會在優先順序清單中選擇下一個節點群組。這在例如您想要使用 P3 執行個體類型的情況下非常有用，因為其 GPU 為您的工作負載提供最佳效能，但做為第二個選項，您也可以使用 P2 執行個體類型。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-autoscaler-priority-expander
  namespace: kube-system
data:
  priorities: |-
    10:
      - .*p2-node-group.*
    50:
      - .*p3-node-group.*
```

Cluster Autoscaler 會嘗試擴展符合 p3-node-group 名稱的 EC2 Auto Scaling 群組。 如果此操作未在 中成功`--max-node-provision-time`，則會嘗試擴展符合名稱 p2-node-group 的 EC2 Auto Scaling 群組。 此值預設為 15 分鐘，並且可以針對更回應的節點群組選擇進行減少，不過如果值太低，可能會導致不必要的橫向擴展。

過度佈建

Cluster Autoscaler 可確保節點只會在需要時新增至叢集，並在未使用時移除，從而將成本降至最低。這將大幅影響部署延遲，因為許多 Pod 會被迫等待節點擴展，才能排程節點。節點可能需要數分鐘才能使用，這樣一來，便會使 Pod 排程延遲增加一個數量級。

這可以使用過度佈建，其會交易排程延遲的成本。過度佈建是使用具有負優先順序的暫時 Pod 實作，這會佔用叢集中的空間。當新建立的 Pod 無法排程且優先順序較高時，臨時 Pod 會先佔空間。然後，暫時 Pod 變得無法排程，觸發 Cluster Autoscaler 擴展新的過度佈建節點。

過度佈建還有其他較不明顯的好處。在不過度佈建的情況下，高度使用叢集的副作用之一，就是 Pod 會使用 Pod 或節點親和`preferredDuringSchedulingIgnoredDuringExecution`性規則做出較不理想的排程決策。常見的使用案例是使用 AntiAffinity，跨可用區域區隔高可用性應用程式的 Pod。過度佈建可能會大幅增加提供正確區域節點的機會。

對於您的組織而言，過度佈建的容量是謹慎的業務決策。其核心是效能和成本之間的權衡。做出此決策的其中一種方法是判斷您的平均擴展頻率，並將其除以擴展新節點所需的時間。例如，如果平均每 30 秒需要新的節點，而 EC2 需要 30 秒來佈建新的節點，則過度佈建的單一節點將確保永遠有可用的額外節點，以單一額外 EC2 執行個體的成本減少 30 秒的排程延遲。為了改善區域排程決策，過度佈建多個節點，等同於 EC2 Auto Scaling 群組中的可用區域數量，以確保排程器可以為傳入的 Pod 選取最佳區域。

防止縮減規模

移出有些工作負載代價高昂。大數據分析、機器學習任務和測試執行器最終將完成，但如果中斷，則必須重新啟動。Cluster Autoscaler 會嘗試縮減 scale-down-utilization-threshold 下的任何節點，這會中斷節點上任何剩餘的 Pod。這可以透過確保昂貴的 Pod 受到 Cluster Autoscaler 識別的標籤保護來防止。

請確認以下事項：

- 昂貴的移出 Pod 具有 註釋 `cluster-autoscaler.kubernetes.io/safe-to-evict=false`

進階使用案例

EBS 磁碟區

持久性儲存對於建置具狀態的應用程式至關重要，例如資料庫或分散式快取。EBS 磁碟區會在 Kubernetes 上啟用此使用案例，但僅限於特定區域。如果針對每個 AZs使用個別的 EBS 磁碟區，在多個 AZ 之間分割，則這些應用程式可以高度使用。然後，Cluster Autoscaler 可以平衡 EC2 Autoscaling 群組的擴展。

請確認以下事項：

- 節點群組平衡是藉由設定 `balance-similar-node-groups=true` 啟用的。

- 除了不同的可用區域和 EBS 磁碟區之外，節點群組的設定相同。

共同排程

機器學習分散式訓練任務可從相同區域節點組態的最低延遲中獲益。這些工作負載會將多個 Pod 部署到特定區域。這可以透過使用 為所有共同排程的 Pod 或節點親和性設定 Pod 親和性來實現`topologyKey: failure-domain.beta.kubernetes.io/zone`。然後，Cluster Autoscaler 會向外擴展特定區域以符合需求。您可能想要配置多個 EC2 Auto Scaling 群組，每個可用區域一個，以啟用整個共同排程工作負載的容錯移轉。

請確認以下事項：

- 節點群組平衡是藉由設定 `balance-similar-node-groups=false` 啟用的。
- 當叢集同時包含區域和區域節點群組時，會使用節點親和性和/或 Pod 先佔。
  - 使用節點親和性來強制或鼓勵區域 Pod 以避免區域節點群組，反之亦然。
  - 如果區域 Pod 排程到區域節點群組，這會導致區域 Pod 的容量不平衡。
  - 如果您的區域工作負載可以容忍中斷和重新定位，請設定 Pod 先佔以啟用區域擴展的 Pod，強制先佔和重新排程較少競爭的區域。

加速器

有些叢集會利用 GPU 等特殊硬體加速器。向外擴展時，加速器裝置外掛程式可能需要幾分鐘的時間來向叢集公告資源。Cluster Autoscaler 已模擬此節點將具有加速器，但直到加速器就緒並更新節點的可用資源為止，無法在節點上排程擱置的 Pod。這可能會導致重複不必要的擴增。

此外，即使加速器未使用，也不會考慮縮減具有加速器和高 CPU 或記憶體使用率的節點。由於加速器的相對成本，這種行為可能很昂貴。相反地，如果節點有未佔用的加速器，Cluster Autoscaler 可以套用特殊規則來考慮縮減規模的節點。

為了確保這些案例的正確行為，您可以在加速器節點上設定 kubelet，以便在節點加入叢集之前標記節點。Cluster Autoscaler 將使用此標籤選擇器來觸發加速器最佳化行為。

請確認以下事項：

- GPU 節點的 Kubelet 已使用 設定 `--node-labels k8s.amazonaws.com/accelerator=$ACCELERATOR_TYPE`
- 具有 Accelerator 的節點遵循上述相同的排程屬性規則。

從 0 擴展

Cluster Autoscaler 能夠在零之間擴展節點群組，進而大幅節省成本。它會檢查其 LaunchConfiguration 或 LaunchTemplate 中指定的 InstanceType，以偵測 Auto Scaling 群組的 CPU、記憶體和 GPU 資源。有些 Pod 需要額外的資源，例如 `WindowsENI`或 `PrivateIPv4Address`或特定的 NodeSelectors 或 Taints，這些資源無法從 LaunchConfiguration 探索。Cluster Autoscaler 可以透過從 EC2 Auto Scaling 群組上的標籤探索它們來考慮這些因素。例如：

```
Key: k8s.io/cluster-autoscaler/node-template/resources/$RESOURCE_NAME
Value: 5
Key: k8s.io/cluster-autoscaler/node-template/label/$LABEL_KEY
Value: $LABEL_VALUE
Key: k8s.io/cluster-autoscaler/node-template/taint/$TAINT_KEY
Value: NoSchedule
```

> **Note**
>
> 請記住，擴展至零時，您的容量會傳回 EC2，且未來可能無法使用。

其他參數

有許多組態選項可用來調整 Cluster Autoscaler 的行為和效能。您可以在 GitHub 上取得完整的參數清單。

參數	描述	預設
掃描間隔	重新評估叢集擴展或縮減的頻率	10 秒
max-empty-bulk-delete	可同時刪除的空節點數量上限。	10
scale-down-delay-after-add	在縮減規模之後，縮減規模評估會繼續多久	10 分鐘
scale-down-delay-after-delete	在縮減評估的節點刪除恢復後多久， 預設為掃描間隔	掃描間隔

參數	描述	預設
scale-down-delay-after-failure	縮減規模失敗後，縮減規模評估會繼續多久	3 分鐘
scale-down-unneeded-time	節點在符合縮減規模的資格之前，應該不需要多長時間	10 分鐘
scale-down-unready-time	不需要尚未就緒的節點多久，才符合縮減規模的資格	20 分鐘
scale-down-utilization-threshold	節點使用率層級，定義為請求的資源總和除以容量，低於此容量時可考慮縮減節點	0.5
scale-down-non-empty-candidates-count	在一次反覆運算中視為具有消耗縮減候選項目的非空節點數目上限。值越低表示 CA 回應能力越好，但縮減延遲可能越慢。較高的值會影響大型叢集（數百個節點）的 CA 效能。設定為非正值以關閉此啟發式 - CA 不會限制其考慮的節點數量。」	30
scale-down-candidates-pool-ratio	當先前反覆運算中的某些候選項目不再有效時，視為額外非空白候選項目的節點比例會縮減。值越低表示 CA 回應能力越好，但縮減延遲可能越慢。較高的值會影響大型叢集（數百個節點）的 CA 效能。設定為 1.0 以關閉此啟發式 - CA 會將所有節點作為其他候選項目。	0.1

參數	描述	預設
scale-down-candidates-pool-min-count	當先前反覆運算中的某些候選項目不再有效時，視為擴展的其他非空白候選項目的節點數量下限。計算其他候選項目的集區大小時 `max(#nodes * scale-down-candidates-pool-ratio, scale-down-candidates-pool-min-count)`	50

其他資源

此頁面包含 Cluster Autoscaler 簡報和示範的清單。如果您想要在此處新增簡報或示範，請傳送提取請求。

簡報/示範	簡報者
Kubernetes 上的自動擴展和成本最佳化：從 0 到 100	Guy 坦伯頓、Skyscanner 和 Jiaxin Shan、Amazon
SIG-Autoscaling Deep Dive	Maciek Pytel 和 Marcin Wielgus

參考

- https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md
- https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/README.md
- https://github.com/aws/amazon-ec2-instance-selector
- https://github.com/aws/aws-node-termination-handler

可靠性的最佳實務

本節提供有關讓工作負載在 EKS 彈性和高可用性上執行的指導

本指南的使用方式

本指南適用於想要在 EKS 中開發和操作高可用性和容錯服務的開發人員和架構師。本指南會組織成不同的主題區域，以便於使用。每個主題都從簡短的概觀開始，後面接著 EKS 叢集可靠性的建議和最佳實務清單。

簡介

EKS 的可靠性最佳實務已分組至下列主題：

- 應用程式
- 控制平面
- 資料平面

是什麼讓系統可靠？ 如果系統可以一致地運作，並在一段時間內滿足環境變更的需求，則可以將其稱為可靠。為了達成此目的，系統必須偵測故障、自動自行修復，並能夠根據需求進行擴展。

客戶可以使用 Kubernetes 作為可靠操作關鍵任務應用程式和服務的基礎。但除了整合容器型應用程式設計原則之外，可靠地執行工作負載也需要可靠的基礎設施。在 Kubernetes 中，基礎設施包含控制平面和資料平面。

EKS 提供生產級 Kubernetes 控制平面，專為高可用性和容錯而設計。

在 EKS 中，AWS 負責 Kubernetes 控制平面的可靠性。EKS 會在 AWS 區域中的三個可用區域執行 Kubernetes 控制平面。它會自動管理 Kubernetes API 伺服器和 等叢集的可用性和可擴展性。

資料平面可靠性的責任由您、客戶和 AWS 共同承擔。EKS 提供四種用於部署 Kubernetes 資料平面的工作者節點選項。

EKS Auto Mode 是最受管的選項，可處理資料平面的佈建、擴展和更新，並提供受管運算、聯網和儲存功能。自動模式 AMIs會經常發行，而叢集會自動更新為最新的 AMI，以部署 CVE 修正和安全性修補程式。您可以在 Auto Mode NodePools 上設定中斷控制，藉此控制何時發生這種情況。

Fargate 透過每個節點執行一個 Pod 來處理資料平面的佈建和擴展。第三個選項是受管節點群組，可處理資料平面的佈建和更新。最後，自我管理節點是資料平面的最低受管選項。您使用的 AWS 受管資料平面越多，您的責任就越少。

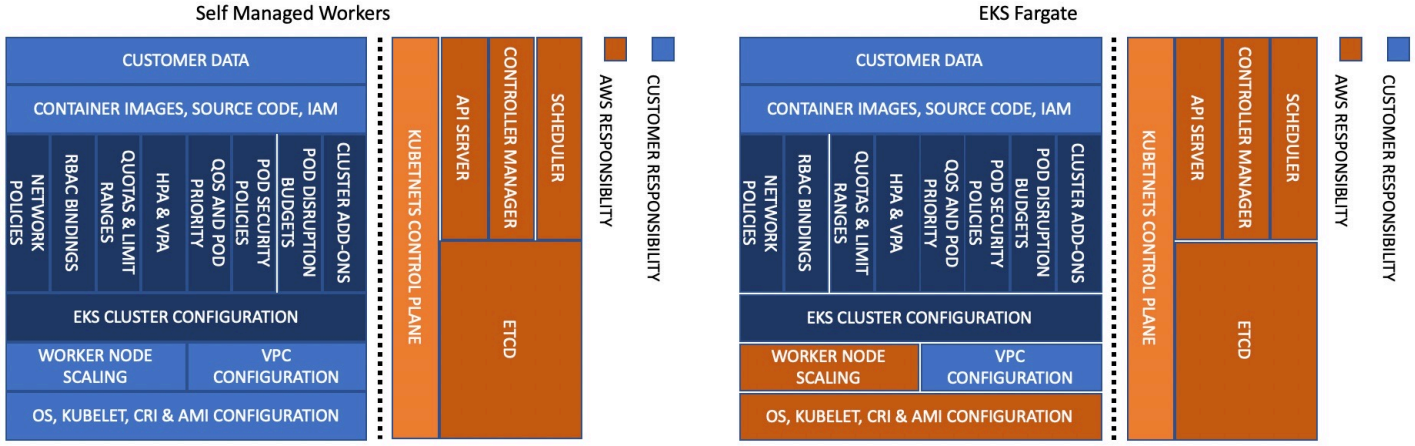
受管節點群組可自動化 EC2 節點的佈建和生命週期管理。您可以使用 EKS API（使用 EKS 主控台、AWS API、AWS CLI、CloudFormation、Terraform 或 `eksctl`) 來建立、擴展和升級受管節點。受管節點會在您的帳戶中執行 EKS 最佳化的 Amazon Linux 2 EC2 執行個體，而且您可以透過啟用 SSH 存取來安裝自訂軟體套件。當您佈建受管節點時，它們會作為 EKS 受管 Auto Scaling 群組的一部分執行，該群組可以跨越多個可用區域；您可以透過您在建立受管節點時提供的子網路來控制。EKS 也會自動標記受管節點，以便與 Cluster Autoscaler 搭配使用。

Amazon EKS 會遵循受管節點群組上 CVE 和安全修補程式的共同責任模型。由於受管節點執行 Amazon EKS 最佳化 AMIs，因此 Amazon EKS 負責在錯誤修正時建置這些 AMIs 的修補版本。不過，您必須負責將這些修補的 AMI 版本部署至受管節點群組。

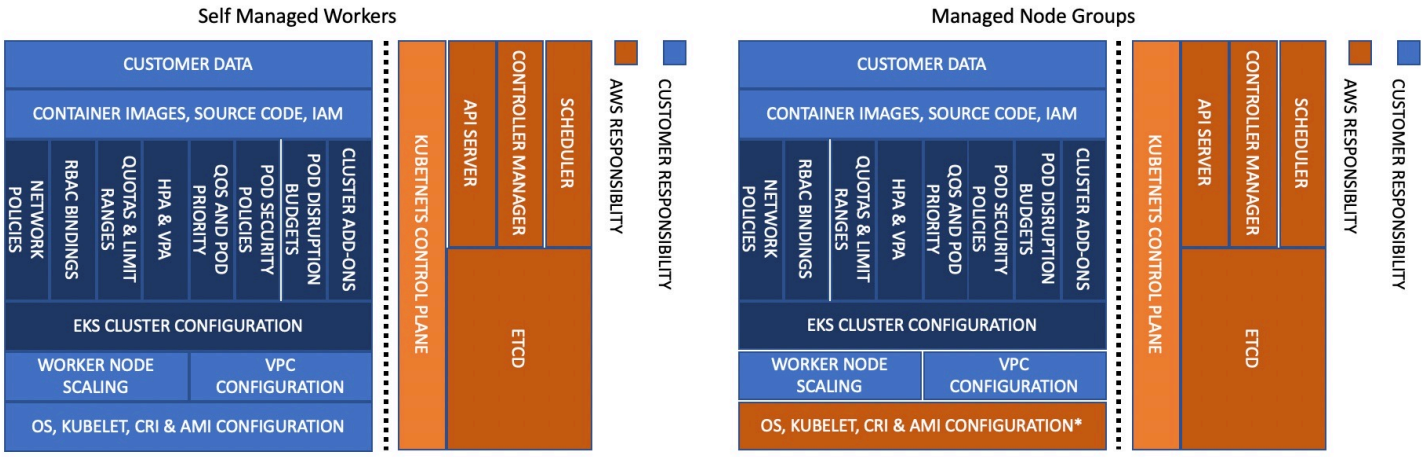
EKS 也會管理更新節點，雖然您必須啟動更新程序。EKS 文件會說明更新受管節點的程序。

如果您執行自我管理節點，您可以使用 Amazon EKS 最佳化 Linux AMI 來建立工作者節點。您負責修補和升級 AMI 和節點。最佳實務是使用 `eksctl`、CloudFormation 或基礎設施做為程式碼工具來佈建自我管理節點，因為這可讓您輕鬆地升級自我管理節點。更新工作者節點時，請考慮遷移至新節點，因為遷移程序會將舊節點群組污點為 `NoSchedule`並在新堆疊準備好接受現有的 Pod 工作負載後耗盡節點。不過，您也可以執行自我管理節點的就地升級。

共同責任模型 - Fargate



共同責任模型 - MNG



本指南包含一組建議，您可以用來改善 EKS 資料平面、Kubernetes 核心元件和應用程式的可靠性。

意見回饋

本指南將在 GitHub 上發佈，以收集更廣泛的 EKS/Kubernetes 社群的直接意見回饋和建議。如果您有最佳實務，認為我們應該包含在指南中，請在 GitHub 儲存庫中提出問題或提交 PR。我們希望在新功能新增至服務或新的最佳實務演進時，定期更新指南。

執行高可用性應用程式

您的客戶預期您的應用程式隨時可用，包括當您進行變更時，特別是在流量激增期間。可擴展且具彈性的架構可讓您的應用程式和服務在不中斷的情況下運作，讓您的使用者滿意。可擴展的基礎設施會根據業務需求成長和縮減。消除單點故障是改善應用程式可用性並提高其彈性的關鍵步驟。

使用 Kubernetes，您可以操作您的應用程式，並以高可用性和彈性的方式執行它們。其宣告式管理可確保設定應用程式後，Kubernetes 會持續嘗試將目前狀態與所需狀態配對。

建議

設定 Pod 中斷預算

Pod 中斷預算用於限制應用程式將遇到的並行中斷量。如果始終有一部分的工作負載可用，則應為工作負載設定它們。EKS Auto Mode、Karpenter 和 Cluster Autoscaler 在縮減規模時，會了解並遵守設定的 Pod 中斷預算。EKS Auto Mode、Karpenter 和受管節點群組在更新節點時也遵循 Pod 中斷預算

避免執行單一 Pod

如果您的整個應用程式在單一 Pod 中執行，則如果該 Pod 終止，您的應用程式將無法使用。建立部署，而不是使用個別 Pod 部署應用程式。如果部署建立的 Pod 失敗或終止，部署控制器會啟動新的 Pod，以確保指定數量的複本 Pod 一律執行。

執行多個複本

使用部署執行應用程式的多個複本 Pod，有助於以高可用性的方式執行。如果一個複本失敗，剩餘的複本仍會正常運作，即使容量減少，直到 Kubernetes 建立另一個 Pod 來彌補遺失為止。此外，您可以使用 Horizontal Pod Autoscaler 根據工作負載需求自動擴展複本。

跨節點排程複本

如果所有複本都在同一個節點上執行，而且節點變得無法使用，則執行多個複本會非常有用。考慮使用 Pod 反親和性或 Pod 拓撲分散限制，將部署的複本分散到多個工作者節點。

您可以透過在多個 AZs 中執行應用程式，進一步改善典型應用程式的可靠性。

使用 Pod 反親和性規則

下列資訊清單會告知 Kubernetes 排程器偏好將 Pod 放置在不同的節點和 AZs 上。它不需要不同的節點或 AZ，因為如果這樣做，則一旦每個 AZ 中有一個執行中的 Pod，Kubernetes 將無法排程任何 Pod。如果您的應用程式只需要三個複本，您可以將 `requiredDuringSchedulingIgnoredDuringExecution`用於 `topologyKey: topology.kubernetes.io/zone`，Kubernetes 排程器不會在相同的可用區域中排程兩個 Pod。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spread-host-az
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
```

```
        app: web-server
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web-server
              topologyKey: topology.kubernetes.io/zone
            weight: 100
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web-server
              topologyKey: kubernetes.io/hostname
            weight: 99
      containers:
      - name: web-app
        image: nginx:1.16-alpine
```

使用 Pod 拓撲分散限制

與 Pod 反親和性規則類似，Pod 拓撲分散限制可讓您在不同故障（或拓撲）網域間使用應用程式，例如主機或可用AZs。當您嘗試在每個不同的拓撲網域中擁有多個複本，以確保容錯能力和可用性時，此方法非常有效。另一方面，Pod 反親和性規則可以輕鬆地產生您在拓撲網域中具有單一複本的結果，因為對彼此具有反親和性的 Pod 具有回推效果。在這種情況下，專用節點上的單一複本不適合容錯能力，也不適合使用 資源。透過拓撲分散限制，您可以更好地控制排程器應嘗試跨拓撲網域套用的分散或分佈。以下是在此方法中使用的一些重要屬性：

1. maxSkew 用於控制或判斷物件在拓撲網域間可能不均勻的最大點。例如，如果應用程式有 10 個複本，並部署在 3 AZs，則無法獲得均勻分散，但可以影響分佈的不均勻程度。在這種情況下， maxSkew可以是介於 1 到 10 之間的任何項目。值 1 表示最終可能會像 一樣分散4,3,3，3,4,3或3,3,4分散到 3 AZs。相反地，值 10 表示最終可能會像 一樣分散10,0,0，0,10,0或0,0,10分散到 3 AZs。

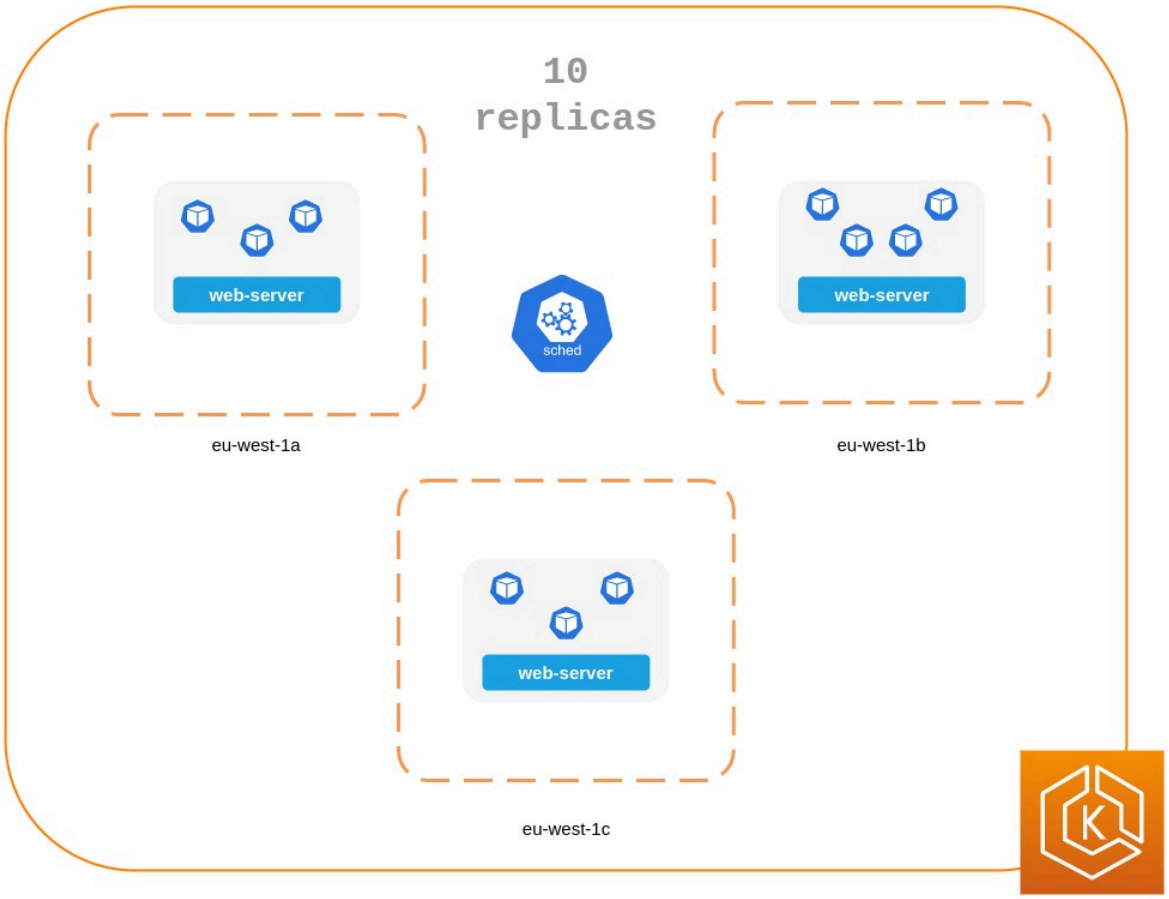
2. `topologyKey` 是其中一個節點標籤的金鑰，並定義應該用於 Pod 分佈的拓撲網域類型。例如，區域分散會有下列索引鍵/值對：

```
topologyKey: "topology.kubernetes.io/zone"
```

3. `whenUnsatisfiable` 如果無法滿足所需的限制條件， 屬性會用來決定您希望排程器如何回應。
4. `labelSelector` 用於尋找相符的 Pod，以便排程器在根據您指定的限制條件決定放置 Pod 的位置時，可以知道它們。

除了上述內容之外，您還可以在 Kubernetes 文件中進一步閱讀其他欄位。

Pod 拓撲將限制分散到 3 AZs




```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: spread-host-az
  labels:
    app: web-server
spec:
  replicas: 10
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: "topology.kubernetes.io/zone"
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            app: express-test
      containers:
      - name: web-app
        image: nginx:1.16-alpine
```

執行 Kubernetes 指標伺服器

安裝 Kubernetes 指標伺服器，以協助擴展您的應用程式。HPA 和 VPA 等 Kubernetes Autoscaler 附加元件需要追蹤應用程式的指標才能進行擴展。指標伺服器會收集可用於進行擴展決策的資源指標。系統會從 kubelet 收集指標，並以指標 API 格式提供。

指標伺服器不會保留任何資料，也不是監控解決方案。其目的是向其他系統公開 CPU 和記憶體用量指標。如果您想要隨著時間追蹤應用程式的狀態，您需要監控工具，例如 Prometheus 或 Amazon CloudWatch。

遵循 EKS 文件，在您的 EKS 叢集中安裝 metrics-server。

水平 Pod Autoscaler (HPA)

HPA 可以自動擴展您的應用程式以回應需求，並協助您避免在尖峰流量期間影響客戶。它在 Kubernetes 中實作為控制迴圈，定期從提供資源指標APIs 查詢指標。

HPA 可以從下列 APIs 擷取指標：1. `metrics.k8s.io`也稱為資源指標 API — 為 Pod 2 提供 CPU 和記憶體用量。 `custom.metrics.k8s.io`— 提供來自 Prometheus 等其他指標收集器的指標；這些指標位於 Kubernetes 叢集內部。3. `external.metrics.k8s.io` — 提供 Kubernetes 叢集外部的指標 （例如 SQS 佇列深度、ELB 延遲）。

您必須使用這三個 APIs之一來提供指標來擴展您的應用程式。

根據自訂或外部指標擴展應用程式

您可以使用自訂或外部指標，根據 CPU 或記憶體使用率以外的指標來擴展應用程式。自訂指標 API 伺服器提供 HPA 可用來自動擴展應用程式的 `custom-metrics.k8s.io` API。

您可以使用 Prometheus Adapter for Kubernetes Metrics APIs 從 Prometheus 收集指標，並搭配 HPA 使用。在此情況下，Prometheus 轉接器將以指標 API 格式公開 Prometheus 指標。

部署 Prometheus Adapter 後，您可以使用 kubectl 查詢自訂指標。 `kubectl get –raw /apis/custom.metrics.k8s.io/v1beta1/`

如名稱所示，外部指標可讓 Horizontal Pod Autoscaler 使用 Kubernetes 叢集外部的指標擴展部署。例如，在批次處理工作負載中，根據 SQS 佇列中正在執行的任務數量來擴展複本數量是很常見的。

若要自動擴展 Kubernetes 工作負載，您可以使用 KEDA (Kubernetes 事件驅動的 Autoscaling)，這是可以根據多個自訂事件驅動容器擴展的開放原始碼專案。此 AWS 部落格概述如何使用 Amazon Managed Service for Prometheus for Kubernetes 工作負載自動擴展。

垂直 Pod Autoscaler (VPA)

VPA 會自動調整 Pod 的 CPU 和記憶體保留，以協助您「正確大小」的應用程式。對於需要垂直擴展的應用程式 - 透過增加資源配置來完成 - 您可以使用 VPA 自動擴展 Pod 複本或提供擴展建議。

如果 VPA 需要擴展應用程式，您的應用程式可能會暫時無法使用，因為 VPA 目前的實作不會對 Pod 執行就地調整；而是會重新建立需要擴展的 Pod。

EKS 文件包含設定 VPA 的逐步解說。

Fairwinds Goldilocks 專案提供儀表板，以視覺化 CPU 和記憶體請求和限制的 VPA 建議。其 VPA 更新模式可讓您根據 VPA 建議自動擴展 Pod。

更新應用程式

現代應用程式需要具有高度穩定性和可用性的快速創新。Kubernetes 為您提供工具來持續更新應用程式，而不會中斷您的客戶。

我們來看看一些最佳實務，可讓您快速部署變更，而不會犧牲可用性。

具有執行轉返的機制

使用復原按鈕可逃避災難。最佳實務是在不同的較低環境（測試或開發環境）中測試部署，然後再更新生產叢集。使用 CI/CD 管道可協助您自動化和測試部署。透過持續部署管道，如果升級發生瑕疵，您可以快速還原至較舊的版本。

您可以使用 部署來更新執行中的應用程式。這通常透過更新容器映像來完成。您可以使用 kubectl 更新部署，如下所示：

```
kubectl --record deployment.apps/nginx-deployment set image nginx-deployment
 nginx=nginx:1.16.1
```

--record 引數會記錄部署的變更，並在您需要執行復原時提供協助。 kubectl rollout history deployment會顯示叢集中部署的記錄變更。您可以使用 轉返變更kubectl rollout undo deployment <DEPLOYMENT_NAME>。

根據預設，當您更新需要重新建立 Pod 的部署時，部署會執行滾動更新。換句話說，Kubernetes 只會在部署中更新部分執行中的 Pod，而不是一次更新所有 Pod。您可以控制 Kubernetes 如何透過 RollingUpdateStrategy 屬性執行滾動更新。

執行部署的滾動更新時，您可以使用 Max Unavailable 屬性來指定更新期間無法使用的 Pod 數量上限。部署的 Max Surge 屬性可讓您設定可透過所需 Pod 數量建立的 Pod 數量上限。

考慮調整max unavailable以確保推展不會中斷您的客戶。例如，Kubernetes max unavailable 預設會設定 25%，這表示如果您有 100 個 Pod，您可能只有 75 個 Pod 在推展期間主動運作。如果您的應用程式需要至少 80 個 Pod，則此推展可能會中斷。反之，您可以將 max unavailable設定為 20%，以確保在整個推出期間至少有 80 個功能 Pod。

使用藍/綠部署

變更本質上具有風險，但無法復原的變更可能具有災難性。變更程序可讓您透過復原有效轉換時間，讓增強功能和實驗更安全。藍/綠部署可讓您在發生錯誤時快速撤回變更。在此部署策略中，您會為新版本建立環境。此環境與目前更新的應用程式版本相同。佈建新環境後，流量會路由至新環境。如果新版本產生所需的結果而不產生錯誤，則舊環境會終止。否則，流量會還原至舊版本。

您可以透過建立與現有版本部署相同的新部署，在 Kubernetes 中執行藍/綠部署。一旦確認新部署中的 Pod 執行時沒有發生錯誤，您可以在 服務中變更將流量路由至應用程式 Pod 的selector規格，以開始將流量傳送至新部署。

許多持續整合工具，例如 Flux、Jenkins 和 Spinnaker，可讓您自動化藍/綠部署。AWS Containers 部落格包含使用 AWS Load Balancer 控制器的逐步解說：使用 AWS Load Balancer 控制器進行藍/綠部署、 Canary 部署和 A/B 測試

使用 Canary 部署

Canary 部署是藍/綠部署的變體，可大幅降低變更的風險。在此部署策略中，您會使用較少的 Pod 與舊部署建立新的部署，並將一小部分的流量轉移到新的部署。如果指標指出新版本的效能與現有版本相同或更好，您會逐步增加新部署的流量，同時向上擴展，直到將所有流量轉移到新部署為止。如果發生問題，您可以將所有流量路由到舊的部署，並停止傳送流量到新的部署。

雖然 Kubernetes 不提供執行 Canary 部署的原生方式，但您可以使用 Flagger 和 Istio 等工具。

運作狀態檢查和自我修復

沒有任何軟體沒有錯誤，但 Kubernetes 可協助您將軟體故障的影響降至最低。在過去，如果應用程式當機，則有人必須透過手動重新啟動應用程式來修復情況。Kubernetes 可讓您偵測 Pod 中的軟體故障，並自動將其取代為新的複本。透過 Kubernetes，您可以監控應用程式的運作狀態，並自動取代運作狀態不佳的執行個體。

Kubernetes 支援三種類型的運作狀態檢查：

1. 活體探查
2. 啟動探查 (Kubernetes 1.16+ 版支援)
3. 準備度探查

Kubernetes 代理程式 Kubelet 負責執行上述所有檢查。Kubelet 可以透過三種方式檢查 Pod 的運作狀態：kubelet 可以在 Pod 的容器內執行 shell 命令、傳送 HTTP GET 請求至其容器，或在指定的連接埠上開啟 TCP 通訊端。

如果您選擇在容器內執行 shell 指令碼`exec`的 型探查，請確保 shell 命令在`timeoutSeconds`值過期之前結束。否則，您的節點將具有 `<defunct>` 程序，導致節點失敗。

建議

使用 Liveness Probe 移除運作狀態不佳的 Pod

活體探查可以偵測程序繼續執行的死結條件，但應用程式變得沒有回應。例如，如果您正在執行在連接埠 80 上接聽的 Web 服務，您可以設定 Liveness 探查，以在 Pod 的連接埠 80 上傳送 HTTP

GET 請求。Kubelet 會定期將 GET 請求傳送至 Pod 並預期回應；如果 Pod 在 200-399 之間回應，則 kubelet 會認為 Pod 運作狀態良好；否則，Pod 會標示為運作狀態不佳。如果 Pod 持續運作狀態檢查失敗，kubelet 會將其終止。

您可以使用 `initialDelaySeconds` 來延遲第一個探查。

使用 Liveness Probe 時，請確保應用程式不會遇到所有 Pod 同時失敗 Liveness Probe 的情況，因為 Kubernetes 會嘗試取代所有 Pod，這會使應用程式離線。此外，Kubernetes 將繼續建立新的 Pod，這也會使 Liveness Probes 失敗，對控制平面造成不必要的壓力。避免將 Liveness Probe 設定為取決於 Pod 外部的因素，例如外部資料庫。換言之，無回應external-to-your-Pod 資料庫不應讓您的 Pod 失敗其即時探查。

Sandor Szücs 的 LIVENESS PROBES 為危險，說明錯誤設定探查可能導致的問題。

將 Startup Probe 用於需要更長的時間才能啟動的應用程式

當您的應用程式需要額外的時間啟動時，您可以使用啟動探查來延遲活體和準備度探查。例如，需要從資料庫補充快取的 Java 應用程式最多可能需要兩分鐘才能完全運作。任何存留期或準備度探查在完全運作前都可能失敗。設定啟動探查將允許 Java 應用程式在執行活體或準備度探查之前正常運作。

在啟動探查成功之前，所有其他探查都會停用。您可以定義 Kubernetes 應等待應用程式啟動的時間上限。如果在設定的時間上限之後，Pod 仍然失敗啟動探查，則會終止，並建立新的 Pod。

啟動探查類似於即時探查 – 如果失敗，則會重新建立 Pod。如同 Ricardo A. 在他的文章 Fantastic Probes 和如何設定它們所述，當應用程式的啟動時間無法預測時，應使用啟動探查。如果您知道您的應用程式需要十秒才能啟動，您應該`initialDelaySeconds`改用 Liveness/Readiness Probe 搭配 。

使用整備探查來偵測部分無法使用

雖然活體探查會偵測透過終止 Pod 解決的應用程式故障（因此，重新啟動應用程式），但準備程度探查會偵測應用程式可能暫時無法使用的情況。在這些情況下，應用程式可能會暫時沒有回應；不過，一旦此操作完成，預期會再次正常運作。

例如，在密集磁碟 I/O 操作期間，應用程式可能暫時無法處理請求。在這裡，終止應用程式的 Pod 不是補救措施；同時，傳送至 Pod 的其他請求可能會失敗。

您可以使用整備探查來偵測應用程式中的暫時無法使用，並停止傳送請求至其 Pod，直到其再次運作為止。與 Liveness Probe 不同，失敗會導致 Pod 重新建立，失敗的準備程度探查表示 Pod 不會收到來自 Kubernetes Service 的任何流量。當整備探查成功時，Pod 會繼續接收來自 Service 的流量。

如同 Liveness Probe，避免設定依賴 Pod 外部資源（例如資料庫）的準備程度探查。以下案例顯示配置不佳的準備度會使應用程式無法運作 - 如果 Pod 的準備度探查在應用程式資料庫無法連線時失敗，其他 Pod 複本也會同時失敗，因為它們共用相同的運作狀態檢查條件。以這種方式設定探查將確保資料庫無法使用時，Pod 的整備探查將會失敗，而 Kubernetes 會停止傳送所有 Pod 的流量。

使用整備探查的副作用是，它們可以增加更新部署所需的時間。除非整備探查成功，否則新複本將不會接收流量；在此之前，舊複本將繼續接收流量。

處理中斷

Pod 的生命週期有限 - 即使您有長時間執行的 Pod，也請務必確保 Pod 在時間到時正確終止。根據您的升級策略，Kubernetes 叢集升級可能需要您建立新的工作者節點，這需要在較新的節點上重新建立所有 Pod。適當的終止處理和 Pod 中斷預算可協助您避免服務中斷，因為 Pod 從較舊的節點移除並在較新的節點上重新建立。

升級工作者節點的首選方法是建立新的工作者節點並終止舊節點。在終止工作者節點之前，您應該`drain`這麼做。當工作者節點耗盡時，會安全地移出其所有 Pod。安全地是這裡的關鍵字；當移出工作者上的 Pod 時，它們不只是傳送`SIGKILL`訊號。相反地，`SIGTERM`訊號會傳送至要移出之 Pod 中每個容器的主要程序 (PID 1)。傳送`SIGTERM`訊號後，Kubernetes 會在傳送`SIGKILL`訊號之前提供一些時間（寬限期）給程序。此寬限期預設為 30 秒；您可以在 kubectl 中使用`grace-period`旗標或在 Podspec `terminationGracePeriodSeconds`中宣告來覆寫預設值。

```
kubectl delete pod <pod name> –grace-period=<seconds>
```

容器的主要程序沒有 PID 1 時很常見。請考慮此以 Python 為基礎的範例容器：

```
$ kubectl exec python-app -it ps
 PID USER TIME COMMAND
 1   root 0:00 {script.sh} /bin/sh ./script.sh
 5   root 0:00 python app.py
```

在此範例中， shell 指令碼會接收 `SIGTERM`，這是主要程序，在此範例中剛好是 Python 應用程式，不會收到`SIGTERM`訊號。當 Pod 終止時，Python 應用程式會突然遭到刪除。您可以透過變更容器`ENTRYPOINT`的 來啟動 Python 應用程式來修復此問題。或者，您可以使用 dumb-init 之類的工具，確保您的應用程式可以處理訊號。

您也可以使用容器掛鉤，在容器啟動或停止時執行指令碼或 HTTP 請求。`PreStop`掛鉤動作會在容器收到`SIGTERM`訊號之前執行，且必須在傳送此訊號之前完成。

該terminationGracePeriodSeconds值會在PreStop掛鉤動作開始執行時套用，而不是在傳送SIGTERM訊號時套用。

建議

使用 Pod 中斷預算保護關鍵工作負載

如果應用程式的複本數量低於宣告的閾值，Pod 中斷預算或 PDB 可以暫時停止移出程序。一旦可用的複本數量超過閾值，移出程序就會繼續。您可以使用 PDB 宣告複本的 minAvailable和 maxUnavailable數目。例如，如果您希望應用程式至少有三個副本可供使用，您可以建立 PDB。

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: my-svc-pdb
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app: my-svc
```

上述 PDB 政策會指示 Kubernetes 停止移出程序，直到有三個或更多複本可用為止。節點耗盡遵守 PodDisruptionBudgets。在 EKS 受管節點群組升級期間，節點會以十五分鐘的逾時耗盡。十五分鐘後，如果未強制更新 (EKS 主控台中的 選項稱為滾動更新 ），則更新會失敗。如果強制更新，則會刪除 Pod。

對於自我管理節點，您也可以使用 AWS Node Termination Handler 等工具，以確保 Kubernetes 控制平面適當回應可能導致 EC2 執行個體無法使用的事件，例如 EC2 維護事件和 EC2 Spot 中斷。它使用 Kubernetes API 來繫結節點，以確保沒有排程新的 Pod，然後耗盡它，終止任何執行中的 Pod。

您可以使用 Pod 反親和性來排程不同節點上的部署 Pod，並避免節點升級期間的 PDB 相關延遲。

練習混沌工程

Chaos Engineering 是在分散式系統上進行實驗的紀律，以建立對系統承受生產環境中亂流條件的能力的信心。

在他的部落格中，Dominik Tornow 解釋 Kubernetes 是一種宣告式系統，其中「使用者向系統提供系統所需狀態的表示。然後，系統會考慮目前狀態和所需狀態，以決定要從目前狀態轉換至所需狀態的命令序列。 「 這表示 Kubernetes 一律會儲存所需的狀態，如果系統偏離，Kubernetes 將採取動作來還

原狀態。例如，如果工作者節點無法使用，Kubernetes 會將 Pod 重新排程到另一個工作者節點。同樣地，如果`replica`當機，部署 Contoller 會建立新的 `replica`。如此一來，Kubernetes 控制器會自動修正故障。

Gremlin 等混沌工程工具可協助您測試 Kubernetes 叢集的彈性，並識別單點故障。在您的叢集（及更高版本）中引入人工混沌的工具可以發現系統弱點、提供識別瓶頸和設定錯誤的機會，以及修正受控環境中的問題。Chaos Engineering 理念倡導打破目標和壓力測試基礎設施，將意外停機時間降至最低。

使用服務網格

您可以使用服務網格來改善應用程式的彈性。服務網格可啟用service-to-service通訊，並提高微服務網路的可觀測性。大多數服務網格產品的運作方式是讓小型網路代理與攔截和檢查應用程式網路流量的每個服務一起執行。您可以將應用程式放在網格中，而無需修改應用程式。使用服務代理的內建功能，您可以讓它產生網路統計資料、建立存取日誌，以及將 HTTP 標頭新增至傳出請求以進行分散式追蹤。

服務網格可協助您透過自動請求重試、逾時、中斷電路和限制速率等功能，讓您的微服務更具彈性。

如果您操作多個叢集，您可以使用服務網格來啟用跨叢集service-to-service通訊。

服務網格

- 伊斯提奧
- LinkerD
- Consul

可觀測性

可觀測性是一個包含監控、記錄和追蹤的概括術語。微服務型應用程式本質上會進行分佈。與監控單一系統的單一應用程式不同，在分散式應用程式架構中，您需要監控每個元件的效能。您可以使用叢集層級監控、記錄和分散式追蹤系統來識別叢集中的問題，然後再中斷您的客戶。

用於疑難排解和監控的 Kubernetes 內建工具有限。指標伺服器會收集資源指標並將其儲存在記憶體中，但不會保留它們。您可以使用 kubectl 檢視 Pod 的日誌，但 Kubernetes 不會自動保留日誌。而分散式追蹤的實作是在應用程式程式碼層級或使用 服務網格來完成。

Kubernetes 的可擴展性在此呈現。Kubernetes 可讓您使用偏好的集中式監控、記錄和追蹤解決方案。

建議

監控您的應用程式

您需要在現代應用程式中監控的指標數量持續增加。如果您有自動追蹤應用程式的方式，可以協助您專注於解決客戶的挑戰。Prometheus 或 CloudWatch Container Insights 等全叢集監控工具可以監控您的叢集和工作負載，並在發生錯誤之前提供訊號給您。

監控工具可讓您建立營運團隊可訂閱的提醒。考慮規則，以針對可能在加劇時導致中斷或影響應用程式效能的事件啟用警示。

如果您不確定應該監控哪些指標，可以從這些方法中獲取啟發：

- RED 方法。代表請求、錯誤和持續時間。
- 使用方法。代表使用率、飽和度和錯誤。

Sysdig 在 Kubernetes 上提醒的後期最佳實務包括可能影響應用程式可用性的元件完整清單。

使用 Prometheus 用戶端程式庫公開應用程式指標

除了監控應用程式的狀態和彙總標準指標之外，您也可以使用 Prometheus 用戶端程式庫公開應用程式特定的自訂指標，以改善應用程式的可觀測性。

使用集中式記錄工具來收集和保留日誌

在 EKS 中記錄分為兩個類別：控制平面日誌和應用程式日誌。EKS 控制平面記錄會將稽核和診斷日誌直接從控制平面提供給帳戶中的 CloudWatch Logs。應用程式日誌是叢集內執行的 Pod 所產生的日誌。應用程式日誌包含執行商業邏輯應用程式和 Kubernetes 系統元件的 Pod 所產生的日誌，例如 CoreDNS、Cluster Autoscaler、Prometheus 等。

EKS 提供五種類型的控制平面日誌：

1. Kubernetes API 伺服器元件日誌
2. 稽核
3. 驗證器
4. 控制器管理員
5. 排程器

控制器管理員和排程器日誌可協助診斷控制平面問題，例如瓶頸和錯誤。根據預設，EKS 控制平面日誌不會傳送至 CloudWatch Logs。您可以啟用控制平面記錄，然後選取您要為帳戶中的每個叢集擷取的 EKS 控制平面日誌類型

收集應用程式日誌需要在您的叢集中安裝日誌彙總工具，例如 Fluent Bit、Fluentd 或 CloudWatch Container Insights。

Kubernetes 日誌彙總工具會以 DaemonSets 的形式執行，並從節點抓取容器日誌。然後，應用程式日誌會傳送到集中式目的地進行儲存。例如，CloudWatch Container Insights 可以使用 Fluent Bit 或 Fluentd 來收集日誌，並將其運送到 CloudWatch Logs 進行儲存。Fluent Bit 和 Fluentd 支援許多熱門日誌分析系統，例如 Elasticsearch 和 InfluxDB，可讓您透過修改 Fluent 位元或 Fluentd 的日誌組態來變更日誌的儲存後端。

使用分散式追蹤系統來識別瓶頸

典型的現代應用程式具有透過網路分佈的元件，其可靠性取決於組成應用程式的每個元件是否正常運作。您可以使用分散式追蹤解決方案來了解請求流程和系統通訊的方式。追蹤可以顯示應用程式網路中存在的瓶頸，並防止可能導致串聯失敗的問題。

您有兩個選項可在應用程式中實作追蹤：您可以使用共用程式庫在程式碼層級實作分散式追蹤，或使用服務網格。

在程式碼層級實作追蹤可能不利。在此方法中，您必須變更程式碼。如果您有多槽應用程式，這會更複雜。您也必須負責維護您服務中的另一個程式庫。

LinkerD 和 Istio 等 Service Meshes 可用於在應用程式中實作分散式追蹤，並將應用程式程式碼的變更降至最低。您可以使用服務網格來標準化指標產生、記錄和追蹤。

Jaeger 可追蹤 AWS X-Ray 等工具，同時支援共用程式庫和服務網格實作。

請考慮使用支援（共用程式庫和服務網格）實作的 AWS X-Ray 或 Jaeger 等追蹤工具，因此如果您稍後採用服務網格，則不需要切換工具。

EKS 控制平面

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

Amazon Elastic Kubernetes Service (EKS) 是一種受管 Kubernetes 服務，可讓您輕鬆地在 AWS 上執行 Kubernetes，而無需安裝、操作和維護您自己的 Kubernetes 控制平面或工作者節點。它執行上游 Kubernetes 並通過 Kubernetes 認證。此一致性可確保 EKS 支援 Kubernetes APIs，就像您可以在 EC2 或內部部署上安裝的開放原始碼社群版本一樣。在上游 Kubernetes 上執行的現有應用程式與 Amazon EKS 相容。

EKS 會自動管理 Kubernetes 控制平面節點的可用性和可擴展性，並自動取代運作狀態不佳的控制平面節點。

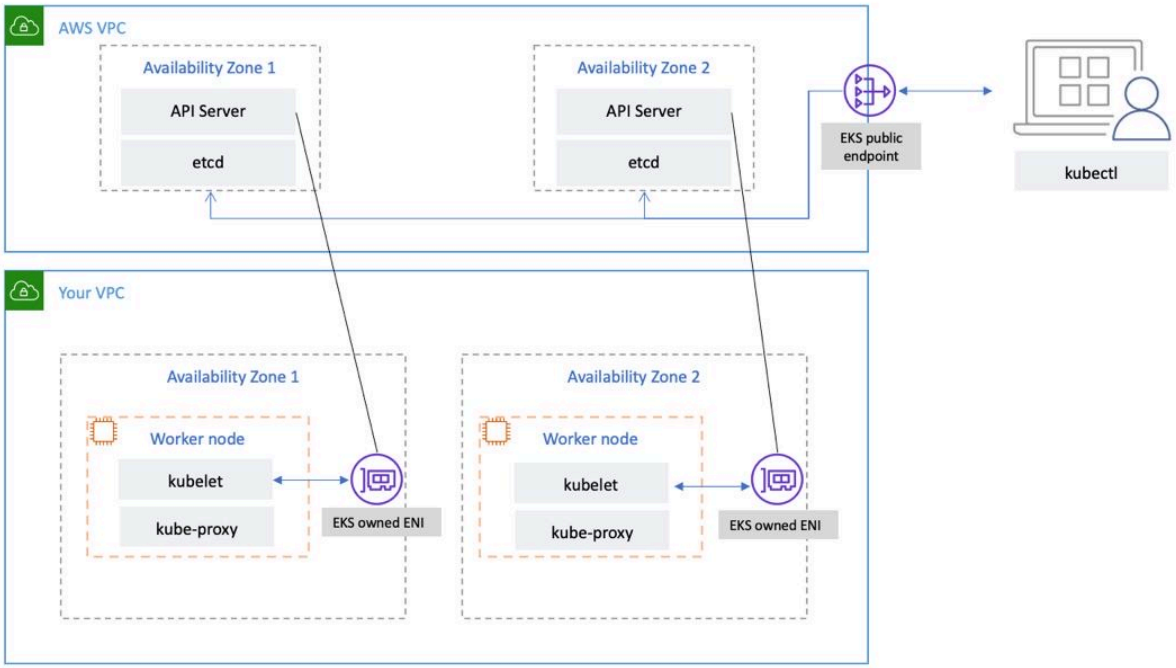
EKS 架構

EKS 架構旨在消除可能影響 Kubernetes 控制平面可用性和耐用性的任何單一故障點。

由 EKS 管理的 Kubernetes 控制平面會在 EKS 受管 VPC 內執行。EKS 控制平面包含 Kubernetes API 伺服器節點等叢集。執行 API 伺服器、排程器等元件並在自動擴展群組中`kube-controller-manager`執行的 Kubernetes API 伺服器節點。EKS 會在 AWS 區域內的不同可用區域 (AZs) 中執行至少兩個 API 伺服器節點。同樣地，為了耐久性，在跨越三個 AZs 的自動調整規模群組中，也會執行 等伺服器節點。EKS 會在每個 AZ 中執行 NAT Gateway，而 API 伺服器和等伺服器會在私有子網路中執行。此架構可確保單一 AZ 中的事件不會影響 EKS 叢集的可用性。

當您建立新的叢集時，Amazon EKS 會為您用來與叢集通訊的受管 Kubernetes API 伺服器建立高可用性端點（使用 等工具`kubectl`)。受管端點使用 NLB 來平衡 Kubernetes API 伺服器。EKS 也會在不同AZs佈建兩個 ENI ，以促進與工作者節點的通訊。

EKS 資料平面網路連線



您可以設定 Kubernetes 叢集的 API 伺服器是否可從公有網際網路（使用公有端點）或透過 VPC（使用 EKS 受管 ENIs) 或兩者連線。

無論使用者和工作者節點使用公有端點或 EKS 受管 ENI 連線至 API 伺服器，都有連線的備援路徑。

建議

檢閱下列建議。

監控控制平面指標

監控 Kubernetes API 指標可讓您深入了解控制平面效能並識別問題。運作狀態不佳的控制平面可能會影響叢集內所執行工作負載的可用性。例如，寫入不良的控制器可能會使 API 伺服器超載，進而影響應用程式的可用性。

Kubernetes 會在`/metrics`端點公開控制平面指標。

您可以使用 檢視公開的指標`kubectl`：

```
kubectl get --raw /metrics
```

這些指標會以 Prometheus 文字格式表示。

您可以使用 Prometheus 來收集和存放這些指標。2020 年 5 月，CloudWatch 新增了在 CloudWatch Container Insights 中監控 Prometheus 指標的支援。因此，您也可以使用 Amazon CloudWatch 來監控 EKS 控制平面。您可以使用教學課程來新增 Prometheus Scrape 目標：Prometheus KPI 伺服器指標來收集指標，並建立 CloudWatch 儀表板來監控叢集的控制平面。

您可以在此處找到 Kubernetes API 伺服器指標。例如，`apiserver_request_duration_seconds`可以指出 API 請求執行所需的時間。

請考慮監控這些控制平面指標：

API 伺服器

指標	說明
`apiserver_request_total`	每個動詞、試轉值、群組、版本、資源、範圍、元件和 HTTP 回應碼的清除請求計數器。
`apiserver_request_duration_seconds*`	每個動詞、幹執行值、群組、版本、資源、子資源、範圍和元件的回應延遲長條圖，以秒為單位。
`apiserver_admission_controller_admission_duration_seconds*`	以秒為單位的許可控制器延遲長條圖，以名稱識別，並針對每個操作和 API 資源和類型（驗證或認可）進行細分。
`apiserver_admission_webhook_rejection_count`	許可 Webhook 拒絕的計數。依名稱、操作、Rejection_code、類型（驗證或許可）、錯誤_類型 (calling_webhook_error、apiserver_internal_error、no_error) 識別
`rest_client_request_duration_seconds*`	請求延遲長條圖，以秒為單位。被動詞和 URL 分解。
`rest_client_requests_total`	HTTP 請求的數量，依狀態碼、方法和主機分割。

- 直方圖指標包括 _bucket、_sum 和 _count 尾碼。

etcd

指標	說明
etcd_request_duration_seconds*	每個操作和物件類型的 Etcd 請求延遲長條圖，以秒為單位。
apiserver_storage_db_total_size_in_bytes 或 apiserver_storage_size_bytes （從 EKS 1.28 版開始）	Etcd 資料庫大小。

- 直方圖指標包括 _bucket、_sum 和 _count 尾碼。

請考慮使用 Kubernetes 監控概觀儀表板來視覺化和監控 Kubernetes API 伺服器請求，以及延遲和等延遲指標。

> **Important**
>
> 當超過資料庫大小限制時， etcd 會發出無空間警示，並停止進一步寫入請求。換言之，叢集會變成唯讀，而所有變更物件的請求，例如建立新 Pod、擴展部署等，都會遭到叢集的 API 伺服器拒絕。

叢集身分驗證

EKS 目前支援兩種類型的身分驗證：承載/服務帳戶字符和使用 Webhook 字符身分驗證的 IAM 身分驗證。當使用者呼叫 Kubernetes API 時，Webhook 會將請求中包含的身分驗證字符傳遞給 IAM。字符是基本 64 簽章的 URL，由 AWS Command Line Interface (AWS CLI) 產生。

建立 EKS 叢集的 IAM 使用者或角色會自動取得叢集的完整存取權。您可以透過編輯 aws-auth 組態對應來管理對 EKS 叢集的存取。

如果您設定錯誤`aws-auth`且無法存取叢集，您仍然可以使用叢集建立者的使用者或角色來存取 EKS 叢集。

萬一您無法在 AWS 區域中使用 IAM 服務，您也可以使用 Kubernetes 服務帳戶的承載字符來管理叢集。

建立允許在叢集中執行所有動作`super-admin`的帳戶：

```
kubectl -n kube-system create serviceaccount super-admin
```

建立提供超級管理員叢集管理員角色的角色繫結：

```
kubectl create clusterrolebinding super-admin-rb --clusterrole=cluster-admin --
serviceaccount=kube-system:super-admin
```

取得服務帳戶的秘密：

```
SECRET_NAME=`kubectl -n kube-system get serviceaccount/super-admin -o
 jsonpath='{.secrets[0].name}'`
```

取得與秘密相關聯的字符：

```
TOKEN=`kubectl -n kube-system get secret $SECRET_NAME -o jsonpath='{.data.token}'|
 base64 --decode`
```

將服務帳戶和字符新增至 kubeconfig：

```
kubectl config set-credentials super-admin --token=$TOKEN
```

將 中的目前內容設定為kubeconfig使用超級管理員帳戶：

```
kubectl config set-context --current --user=super-admin
```

最終kubeconfig應如下所示：

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data:<REDACTED>
    server: https://<CLUSTER>.gr7.us-west-2.eks.amazonaws.com
  name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
contexts:
- context:
    cluster: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
    user: super-admin
  name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
current-context: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
kind: Config
preferences: {}
users:
#- name: arn:aws:eks:us-west-2:<account number>:cluster/<cluster name>
#  user:
#    exec:
#      apiVersion: client.authentication.k8s.io/v1beta1
#      args:
#      - --region
#      - us-west-2
#      - eks
```

```
#      - get-token
#      - --cluster-name
#      - <<cluster name>>
#      command: aws
#      env: null
- name: super-admin
  user:
    token: <<super-admin sa's secret>>
```

許可 Webhook

Kubernetes 有兩種類型的許可 Webhook：驗證許可 Webhook 和變更許可 Webhook。這些允許使用者擴展 kubernetes API，並在 API 接受物件之前對其進行驗證或變更。這些 Webhook 的組態不佳，可以透過封鎖叢集關鍵操作來取消穩定 EKS 控制平面。

為了避免影響叢集關鍵操作，請避免設定如下所示的「全部擷取」 Webhook：

```
- name: "pod-policy.example.com"
  rules:
  - apiGroups:   ["*"]
    apiVersions: ["*"]
    operations:  ["*"]
    resources:   ["*"]
    scope: "*"
```

或者，確定 Webhook 具有故障開啟政策，其逾時短於 30 秒，以確保如果 Webhook 無法使用，這不會影響叢集關鍵工作負載。

封鎖具有不安全的 Pod **sysctls**

Sysctl 是一種 Linux 公用程式，可讓使用者在執行時間修改核心參數。這些核心參數可控制作業系統行為的各個層面，例如網路、檔案系統、虛擬記憶體和程序管理。

Kubernetes 允許指派 Pod 的sysctl設定檔。Kubernetes 分類systcls為安全和不安全。安全sysctls在容器或 Pod 中命名空間，設定它們不會影響節點或節點本身的其他 Pod。相反地，不安全的 sysctl 預設為停用，因為它們可能會中斷其他 Pod 或使節點不穩定。

由於預設為sysctls停用不安全，因此 kubelet 不會建立具有不安全sysctl設定檔的 Pod。如果您建立這類 Pod，排程器會重複將這類 Pod 指派給節點，而節點無法啟動。此無限迴圈最終會對叢集控制平面造成壓力，使叢集不穩定。

考慮使用 OPA Gatekeeper 或 Kyverno 來拒絕具有不安全 的 Podsysctls。

處理叢集升級

自 2021 年 4 月起，Kubernetes 發行週期已從每年四個版本 （每季一次） 變更為每年三個版本。新的次要版本 （例如 1.21 或 1.22) 大約每 15 週發行一次。從 Kubernetes 1.19 開始，每個次要版本在第一次發行後支援大約 12 個月。隨著 Kubernetes v1.28 的推出，控制平面和工作者節點之間的相容性偏移已從 n-2 擴展到 n-3 次要版本。若要進一步了解，請參閱叢集升級的最佳實務。

叢集端點連線能力

使用 Amazon EKS （彈性 Kubernetes Service) 時，您可能會在 Kubernetes 控制平面擴展或修補等事件期間遇到連線逾時或錯誤。這些事件可能會導致 kube-apiserver 執行個體遭到取代，可能會導致解析 FQDN 時傳回不同的 IP 地址。本文件概述 Kubernetes API 消費者維持可靠連線的最佳實務。

> **Note**
>
> 實作這些最佳實務可能需要更新用戶端組態或指令碼，才能有效地處理新的 DNS 重新解析和重試策略。

主要問題來自 DNS 用戶端快取，以及 EKS 端點 - 公有端點的公有 NLB 或私有端點的 X-ENI 的可能過時 IP 地址。取代 kube-apiserver 執行個體時，完整網域名稱 (FQDN) 可能會解析為新的 IP 地址。不過，由於 DNS 存留時間 (TTL) 設定在 AWS 受管 Route 53 區域中設定為 60 秒，用戶端可能會在短時間內繼續使用過期的 IP 地址。

為了緩解這些問題，Kubernetes API 取用者 （例如 kubectl、CI/CD 管道和自訂應用程式） 應實作下列最佳實務：

- 實作 DNS 重新解析
- 使用退避和抖動實作重試。例如，請參閱這篇名為 Failures Happen 的文章
- 實作用戶端逾時。設定適當的逾時，以防止長時間執行的請求封鎖您的應用程式。請注意，某些 Kubernetes 用戶端程式庫，特別是 OpenAPI 產生器所產生的程式庫，可能不允許輕鬆設定自訂逾時。
  - 範例 1 搭配 kubectl：

```
kubectl get pods --request-timeout 10s # default: no timeout
```

- 使用 Python 的範例 2：Kubernetes 用戶端提供 _request_timeout 參數

透過實作這些最佳實務，您可以在與 Kubernetes API 互動時大幅改善應用程式的可靠性和彈性。請記得徹底測試這些實作，尤其是在模擬失敗條件下，以確保它們在實際擴展或修補事件期間如預期般運作。

執行大型叢集

EKS 會主動監控控制平面執行個體上的負載，並自動進行擴展以確保高效能。不過，在執行大型叢集時，您應該考慮 Kubernetes 內的潛在效能問題和限制，以及 AWS 服務中的配額。

- 根據 ProjectCalico 團隊執行的測試，使用 `kube-proxy` 以 `iptables` 模式執行超過 1000 個服務的叢集可能會遇到網路延遲。解決方案是切換到`kube-proxy`以 `ipvs` 模式執行。
- 如果 CNI 需要請求 Pod 的 IP 地址，或者如果您需要經常建立新的 EC2 執行個體，您可能也會遇到 EC2 API 請求限流。 EC2 您可以透過設定 CNI 快取 IP 地址來減少呼叫 EC2 API。您可以使用較大的 EC2 執行個體類型來減少 EC2 擴展事件。

其他資源：

- 解密 Amazon EKS 工作者節點的叢集聯網
- Amazon EKS 叢集端點存取控制
- AWS re：Invent 2019：在幕後使用 Amazon EKS (CON421-R1)

EKS 資料平面

若要操作高可用性和彈性應用程式，您需要高可用性和彈性的資料平面。彈性資料平面可確保 Kubernetes 可以自動擴展和修復您的應用程式。彈性資料平面由兩個或多個工作者節點組成，可以隨著工作負載成長和縮減，並從故障中自動復原。

對於具有 EKS 的工作者節點，您有多種選擇：EKS Auto Mode 受管節點、EC2 執行個體和 Fargate。

EKS Auto Mode 提供彈性資料平面的最簡單路徑。Auto Mode 會將 Kubernetes 叢集的 AWS 管理延伸到叢集本身之外，讓 AWS 也可以設定和管理基礎設施，讓工作負載能夠順暢地運作。自動模式會在 Kubernetes 擴展 Pod 時自動擴展或縮減資料平面，並持續確保叢集中的節點大小適合目前執行中的工作負載，且符合成本效益。

如果您選擇 EC2 執行個體，您可以自行管理工作者節點，或使用 EKS 受管節點群組。您可以擁有混合自動模式、受管、自我管理的工作者節點和 Fargate 的叢集。

Fargate 會在隔離的運算環境中執行每個 Pod。在 Fargate 上執行的每個 Pod 都會取得自己的工作者節點。Fargate 會在 Kubernetes 擴展 Pod 時自動擴展資料平面。您可以使用水平 Pod 自動擴展器來擴展資料平面和工作負載。

擴展 EC2 工作者節點（如果未使用 AWS 自動執行的 EKS Auto 模式）的偏好方法是使用 Karpenter、Kubernetes Cluster Autoscaler 或 EC2 Auto Scaling 群組。

建議

將工作者節點和工作負載分散到多個可用AZs

您可以在多個 AZ 中執行工作者節點和 Pod，以保護工作負載免於個別 AZs 中的故障。您可以使用您建立節點的子網路，控制在 中建立工作者節點的 AZ。

跨 AZs 分散 Pod 的建議方法是使用 Pod 的拓撲分散限制條件。EKS Auto Mode 和 Karpenter 等自動擴展功能知道拓撲分散限制條件，並會自動在正確的 AZs 中啟動節點，以滿足您的限制條件。

下列部署會盡可能將 Pod 分散到 AZs，如果沒有的話，讓這些 Pod 執行：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      topologySpreadConstraints:
        - maxSkew: 1
          whenUnsatisfiable: ScheduleAnyway
          topologyKey: topology.kubernetes.io/zone
          labelSelector:
            matchLabels:
```

```
            app: web-server
    containers:
    - name: web-app
      image: nginx
      resources:
        requests:
          cpu: 1
```

> **Note**
>
> kube-scheduler 只會透過與這些標籤一起存在的節點來了解拓撲網域。如果上述部署僅部署到單一區域中具有節點的叢集，則所有 Pod 都會在這些節點上排程，因為 kube-scheduler 不知道其他區域。若要讓此拓撲與排程器如預期般運作，節點必須已存在於所有區域中。拓撲分散限制條件的 minDomains 屬性用於通知排程器合格網域的數量，即使其中有節點執行以避免此問題。

> **Warning**
>
> 如果無法滿足拓撲分散限制條件，whenUnsatisfiableDoNotSchedule將導致 Pod 無法排程。只有在 Pod 不執行而不是違反拓撲分散限制條件時，才應該設定它。

在舊版 Kubernetes 上，您可以使用 Pod 反親和性規則來跨多個 AZs 排程 Pod。下列資訊清單會通知 Kubernetes 排程器偏好在不同的AZs排程 Pod。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
  labels:
    app: web-server
spec:
  replicas: 4
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
```

```
        app: web-server
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - web-server
              topologyKey: failure-domain.beta.kubernetes.io/zone
            weight: 100
      containers:
      - name: web-app
        image: nginx
```

> **Warning**
>
> 不需要在不同的 AZs 之間排程 Pod，否則部署中的 Pod 數目永遠不會超過 AZs 數目。

使用 EBS 磁碟區時，確保能夠在每個 AZ 中啟動節點

如果您使用 Amazon EBS 提供持久性磁碟區，則需要確保 Pod 和相關聯的 EBS 磁碟區位於相同的可用區域中。Pod 無法存取位於不同 AZ 的 EBS 支援的持久性磁碟區。Kubernetes 排程器會從節點上的標籤知道工作者節點位於哪個 AZ，並且一律會排程 Pod，該 Pod 需要與磁碟區位於相同 AZ 中的 EBS 磁碟區。不過，如果磁碟區所在的 AZ 中沒有可用的工作者節點，則無法排程 Pod。

如果使用 EKS Auto Mode 或 Karpenter，您將需要確保 NodeClass 在每個可用區域中選取子網路。如果使用受管節點群組，您需要確保在每個 AZ 中有一個節點群組。

EBS 儲存功能內建於 EKS Auto 模式，但如果使用 Karpenter 或受管節點群組，則還需要安裝 EBS CSI。

使用 EKS Auto 模式管理工作者節點

EKS Auto Mode 透過為生產就緒的叢集提供最低的操作負荷，簡化 EKS 管理。自動模式負責根據叢集中執行的 Pod 向上或向下擴展節點數量。節點會與軟體修補程式保持最新狀態，並自動修正，並根據設定的 NodePool 中斷設定和 Pod 中斷預算執行更新。

執行節點監控代理程式

節點監控代理程式會透過發佈 Kubernetes 事件並更新節點上的狀態條件，來監控節點運作狀態問題並做出反應。節點監控代理程式隨附於 EKS 自動模式節點，可安裝為非自動模式管理之節點的 EKS 附加元件。

EKS Auto Mode、受管節點群組和 Karpenter 全都能夠偵測節點監控代理程式回報的嚴重節點條件，並在這些條件發生時自動修復這些節點。

實作 QoS

對於關鍵應用程式，請考慮為 Pod 中的容器定義 `requests=limits`。這將確保在另一個 Pod 請求資源時不會終止容器。

最佳實務是為所有容器實作 CPU 和記憶體限制，因為它可防止容器無意中耗用系統資源，進而影響其他共置程序的可用性。

設定和調整所有工作負載的資源請求/限制

某些一般指引可以套用到調整工作負載的資源請求和限制大小：

- 請勿在 CPU 上指定資源限制。如果沒有限制，請求會做為容器取得多少相對 CPU 時間的權重。這可讓您的工作負載使用完整的 CPU，而不需要人工限制或匱乏。
- 對於非 CPU 資源，設定 `requests=limits` 提供最可預測的行為。如果 `requests！=limits`，容器也會將其 QOS 從保證縮減為爆量，使其在節點壓力的情況下更有可能被移出。
- 對於非 CPU 資源，請勿指定遠大於請求的限制。相對於 ，`limits`設定較大的節點`requests`越有可能過度承諾，導致工作負載中斷的機率很高。
- 正確大小的請求在使用 Karpenter 或 Cluster AutoScaler 等節點自動擴展解決方案時特別重要。這些工具會查看您的工作負載請求，以決定要佈建的節點數量和大小。如果您的請求太小，且限制較大，您可能會發現工作負載在節點上封裝緊密時遭到移出或 OOM 終止。

判斷資源請求可能很困難，但 Vertical Pod Autoscaler 等工具可以透過在執行時間觀察容器資源用量來協助您「正確大小」請求。其他可能有助於判斷請求大小的工具包括：

- 金絲雀
- Parca
- Prodfiler
- rsg

設定命名空間的資源配額

命名空間主要使用於有許多使用者分散於多個團隊或專案的環境中。它們提供名稱的範圍，並且是在多個團隊、專案、工作負載之間劃分叢集資源的一種方式。您可以限制命名空間中的彙總資源耗用。`ResourceQuota` 物件可以依類型限制可在命名空間中建立的物件數量，以及該專案中資源可能耗用的運算資源總量。您可以限制可在指定命名空間中請求的儲存和/或運算 (CPU 和記憶體） 資源總和。

如果為 CPU 和記憶體等運算資源的命名空間啟用資源配額，使用者必須為該命名空間中的每個容器指定請求或限制。

考慮為每個命名空間設定配額。考慮使用 `LimitRanges` 自動將預先設定的限制套用至命名空間內的容器。

限制命名空間中的容器資源用量

Resource Quotas 有助於限制命名空間可以使用的資源量。`LimitRange` 物件可協助您實作容器可請求的最小和最大資源。`LimitRange` 您可以使用 設定容器的預設請求和限制，如果設定運算資源限制不是組織中的標準實務，這很有幫助。顧名思義， `LimitRange`可以強制執行命名空間中每個 Pod 或容器的運算資源用量下限和上限。此外，在命名空間中強制執行每個 PersistentVolumeClaim 的儲存請求下限和上限。

考慮將 `LimitRange` 與 搭配使用，`ResourceQuota`以在容器和命名空間層級強制執行限制。設定這些限制可確保容器或命名空間不會影響叢集中其他租用戶使用的資源。

使用 NodeLocal DNSCache

您可以執行 NodeLocal DNSCache 來改善叢集 DNS 效能。此功能會在叢集節點上執行 DNS 快取代理程式，做為 DaemonSet。所有 Pod 都會使用節點上執行的 DNS 快取代理程式進行名稱解析，而不是使用 `kube-dns` Service。此功能會自動包含在 EKS Auto 模式中。

設定自動調整規模的 CoreDNS

改善叢集 DNS 效能的另一個方法是啟用 CoreDNS Pod 的內建自動擴展。

此功能會持續監控叢集狀態，包括節點和 CPU 核心的數量。根據該資訊，控制器會動態調整 EKS 叢集中 CoreDNS 部署的複本數量。

聯網的最佳實務

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

請務必了解 Kubernetes 聯網，才能有效率地操作叢集和應用程式。Pod 聯網也稱為叢集聯網，是 Kubernetes 聯網的中心。Kubernetes 支援叢集聯網的容器網路界面 (CNI) 外掛程式。

Amazon EKS 正式支援 Amazon Virtual Private Cloud (VPC) CNI 外掛程式，以實作 Kubernetes Pod 網路。VPC CNI 提供與 AWS VPC 的原生整合，並在底層模式中運作。在底層模式中，Pod 和主機位於相同的網路層，並共用網路命名空間。從叢集和 VPC 的觀點來看，Pod 的 IP 地址是一致的。

本指南介紹 Kubernetes 叢集聯網內容中的 Amazon VPC 容器網路界面 (VPC CNI)。VPC CNI 是 EKS 支援的預設聯網外掛程式，因此是指南的重點。VPC CNI 可高度設定，以支援不同的使用案例。本指南進一步包含不同 VPC CNI 使用案例、操作模式、子元件的專用區段，後面接著建議。

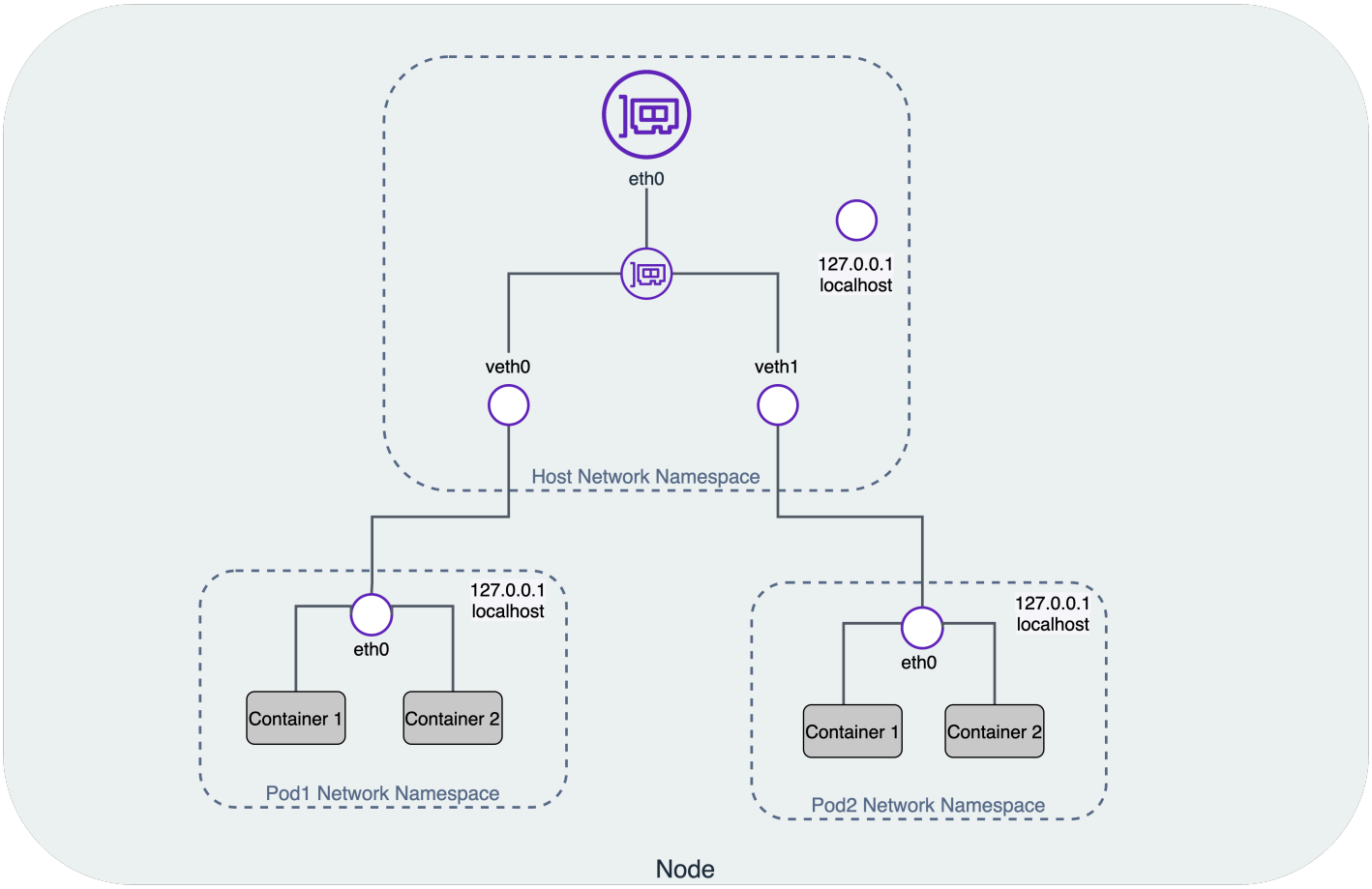
Amazon EKS 執行上游 Kubernetes 並通過 Kubernetes 認證。雖然您可以使用替代 CNI 外掛程式，但本指南不提供管理替代 CNIs 的建議。請查看 EKS 替代 CNI 文件，以取得有效管理替代 CNIs的合作夥伴和資源清單。

Kubernetes 網路模型

Kubernetes 會在叢集聯網上設定下列需求：

- 在同一節點上排程的 Pod 必須能夠與其他 Pod 通訊，而無需使用 NAT （網路位址轉譯）。
- 在特定節點上執行的所有系統精靈 （後端程序，例如 kubelet) 都可以與在相同節點上執行的 Pod 通訊。
- 使用主機網路的 Pod 必須能夠聯絡所有其他節點上的所有其他 Pod，而無需使用 NAT。

如需 Kubernetes 對相容聯網實作有何期望的詳細資訊，請參閱 Kubernetes 網路模型。下圖說明 Pod 網路命名空間與主機網路命名空間之間的關係。



容器聯網界面 (CNI)

Kubernetes 支援 CNI 規格和外掛程式，以實作 Kubernetes 網路模型。CNI 包含規格 （目前版本 1.0.0) 和程式庫，用於撰寫外掛程式以設定容器中的網路介面，以及一些支援的外掛程式。CNI 只擔心容器的網路連線，並在刪除容器時移除配置的資源。

透過傳遞`--network-plugin=cni`命令列選項 kubelet 來啟用 CNI 外掛程式。Kubelet 會從 讀取檔案`--cni-conf-dir`（預設 /etc/cni/net.d)，並使用該檔案中的 CNI 組態來設定每個 Pod 的網路。CNI 組態檔案必須符合 CNI 規格 （最低 v0.4.0)，且組態參考的任何必要 CNI 外掛程式都必須存在於 `--cni-bin-dir`目錄中 （預設 /opt/cni/bin)。如果目錄中有多個 CNI 組態檔案，則 kubelet 會使用依詞彙順序名稱排列的組態檔案。

Amazon Virtual Private Cloud (VPC) CNI

AWS 提供的 VPC CNI 是 EKS 叢集的預設聯網附加元件。佈建 EKS 叢集時，預設會安裝 VPC CNI 附加元件。VPC CNI 在 Kubernetes 工作者節點上執行。VPC CNI 附加元件包含 CNI 二進位檔和

IP 地址管理 (ipamd) 外掛程式。CNI 會將來自 VPC 網路的 IP 地址指派給 Pod。ipamd 會管理每個 Kubernetes 節點的 AWS Elastic Networking Interfaces (ENIs)，並維護 IPs的暖集區。VPC CNI 提供預先配置 ENIs和 IP 地址的組態選項，以加快 Pod 啟動時間。如需建議的外掛程式管理最佳實務，請參閱 Amazon VPC CNI。

Amazon EKS 建議您在建立叢集時，在至少兩個可用區域中指定子網路。Amazon VPC CNI 會從節點子網路將 IP 地址配置給 Pod。我們強烈建議檢查子網路是否有可用的 IP 地址。部署 EKS 叢集之前，請考慮 VPC 和子網路建議。

Amazon VPC CNI 會從連接至節點主要 ENIs 的子網路配置 ENI 和次要 IP 地址的暖集區。此 VPC CNI 模式稱為次要 IP 模式。IP 地址數量和因此 Pod 數量 (Pod 密度） 是由執行個體類型定義的 ENIs 數量和每個 ENI 的 IP 地址 （限制） 所定義。次要模式是預設值，適用於執行個體類型較小的小型叢集。如果您遇到 Pod 密度挑戰，請考慮使用字首模式。您也可以透過將字首指派給 ENIs 來增加 Pod 節點上的可用 IP 地址。

Amazon VPC CNI 原生與 AWS VPC 整合，並允許使用者套用現有的 AWS VPC 網路和安全性最佳實務，以建置 Kubernetes 叢集。這包括使用 VPC 流程日誌、VPC 路由政策和安全群組進行網路流量隔離的能力。根據預設，Amazon VPC CNI 會將與節點上主要 ENI 相關聯的安全群組套用至 Pod。如果您想要為 Pod 指派不同的網路規則，請考慮啟用 Pod 的安全群組。

根據預設，VPC CNI 會將 IP 地址從指派給節點主要 ENI 的子網路指派給 Pod。執行具有數千個工作負載的大型叢集時，通常會遇到 IPv4 地址不足的情況。AWS VPC 可讓您透過指派次要 CIDRs 來擴展可用的 IPs，以解決 IPv4 CIDR 區塊耗盡的問題。AWS VPC CNI 可讓您為 Pod 使用不同的子網路 CIDR 範圍。VPC CNI 的此功能稱為自訂聯網。您可以考慮使用自訂聯網搭配 EKS `100.64.0.0/10`範圍 （共用地址空間 RFC 6598) 的 CIDR。這實際上可讓您建立 Pod 不再從您的 VPC 使用任何 RFC1918 IP 地址的環境。

自訂聯網是解決 IPv4 地址耗盡問題的一種選項，但需要營運開銷。我們建議透過自訂聯網使用 IPv6 叢集來解決此問題。具體而言，如果您已完全耗盡 VPC 的所有可用 IPv4 地址空間，建議您遷移至 IPv6 叢集。評估您組織的計劃以支援 IPv6，並考慮投資 IPv6 是否有更長期的價值。

EKS 對 IPv6 的支援著重於解決有限 IPv4 地址空間所造成的 IP 耗盡問題。為了回應 IPv4 耗盡的客戶問題，EKS 已將IPv6-only 的 Pod 優先於雙堆疊 Pod。也就是說，Pod 可以存取 IPv4 資源，但不會從 VPC CIDR 範圍指派 IPv4 地址。VPC CNI 會從 AWS 受管 VPC IPv6 CIDR 區塊將 IPv6 地址指派給 Pod。

子網路計算器

此專案包含子網路計算器 Excel 文件。此計算器文件會模擬不同 ENI 組態選項下指定工作負載的 IP 地址使用量，例如 WARM_IP_TARGET和 WARM_ENI_TARGET。文件包含兩張工作表，一張用於暖 ENI 模式，另一張用於暖 IP 模式。如需這些模式的詳細資訊，請參閱 VPC CNI 指引。

輸入：

- 子網路 CIDR 大小
- 暖 ENI 目標或暖 IP 目標
- 執行個體清單
  - 每個執行個體排程的工作負載 Pod 類型、數量和數量

輸出：

- 託管的 Pod 總數
- 使用的子網路 IPs 數量
- 剩餘的子網路 IPs 數量
- 執行個體層級詳細資訊
  - 每個執行個體IPs/ENIs數量
  - 每個執行個體IPs/ENIs數量

VPC 和子網路考量事項

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

除了 Kubernetes 聯網之外，操作 EKS 叢集還需要 AWS VPC 聯網的知識。

我們建議您在開始設計 VPC 或將叢集部署到現有 VPCs 之前，先了解 EKS 控制平面通訊機制。

建構要與 EKS 搭配使用的 VPC 和子網路時，請參閱叢集 VPC 考量事項和 Amazon EKS 安全群組考量事項。 https://docs.aws.amazon.com/eks/latest/userguide/sec-group-reqs.html

概觀

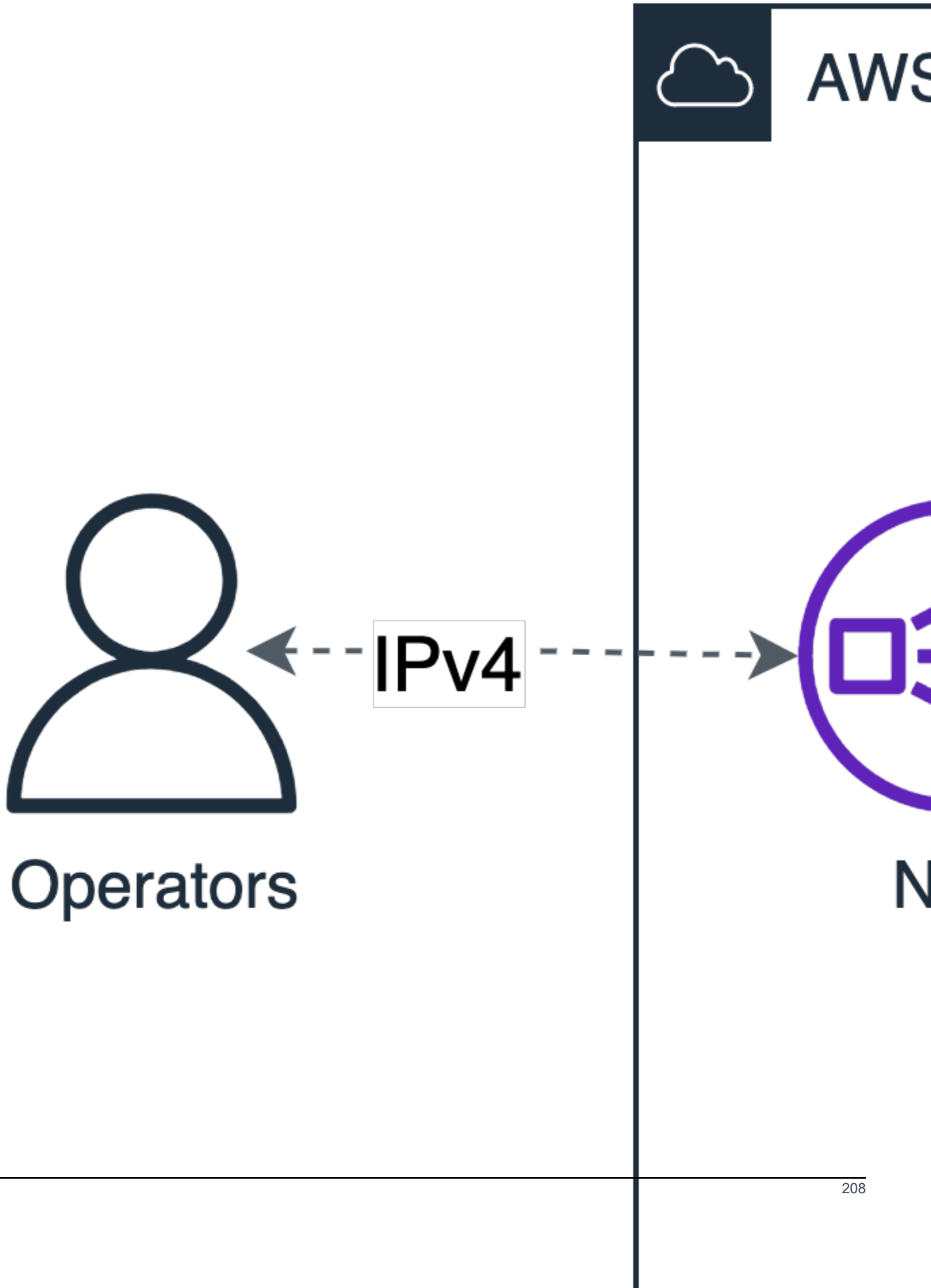
EKS 叢集架構

EKS 叢集包含兩個 VPCs：

- 託管 Kubernetes 控制平面的 AWS 受管 VPC。此 VPC 不會出現在客戶帳戶中。
- 託管 Kubernetes 節點的客戶受管 VPC。這是容器執行的位置，以及其他客戶管理的 AWS 基礎設施，例如叢集所使用的負載平衡器。此 VPC 會出現在客戶帳戶中。您需要先建立客戶受管 VPC，才能建立叢集。如果您不提供 VPC，eksctl 會建立 VPC。

客戶 VPC 中的節點需要連線至 AWS VPC 中受管 API 伺服器端點的能力。這可讓節點向 Kubernetes 控制平面註冊，並接收執行應用程式 Pod 的請求。

節點透過 (a) EKS 公有端點或 (b) 由 EKS 管理的跨帳戶彈性網路介面 (X-ENI) 連接到 EKS 控制平面。建立叢集時，您需要指定至少兩個 VPC 子網路。EKS 會在叢集建立期間指定的每個子網路 （也稱為叢集子網路） 中放置 X-ENI。Kubernetes API 伺服器使用這些跨帳戶 ENIs 與客戶受管叢集 VPC 子網路上部署的節點進行通訊。


AWS
IPv4
Operators
N

當節點啟動時，會執行 EKS 引導指令碼並安裝 Kubernetes 節點組態檔案。做為每個執行個體上開機程序的一部分，容器執行期代理程式、kubelet 和 Kubernetes 節點代理程式都會啟動。

若要註冊節點，Kubelet 會聯絡 Kubernetes 叢集端點。它與 VPC 外部的公有端點或 VPC 內的私有端點建立連線。Kubelet 會收到 API 指示，並定期向端點提供狀態更新和活動訊號。

EKS 控制平面通訊

EKS 有兩種方法來控制對叢集端點的存取。端點存取控制可讓您選擇是否可從公有網際網路或只能透過 VPC 到達端點。您可以一次開啟公有端點（這是預設值）、私有端點或兩者。

叢集 API 端點的組態會決定節點與控制平面通訊所需的路徑。請注意，您可以隨時透過 EKS 主控台或 API 變更這些端點設定。

公有端點

這是新 Amazon EKS 叢集的預設行為。僅啟用叢集的公有端點時，源自叢集 VPC（例如控制平面通訊的工作者節點）的 Kubernetes API 請求會離開 VPC，但不會離開 Amazon 的網路。為了讓節點連接到控制平面，它們必須具有公有 IP 地址和網際網路閘道的路由，或 NAT 閘道的路由，他們可以使用 NAT 閘道的公有 IP 地址。

公有和私有端點

同時啟用公有和私有端點時，來自 VPC 內部的 Kubernetes API 請求會透過 VPC 中的 X-ENIs 與控制平面通訊。您的叢集 API 伺服器可從網際網路上存取。

私有端點

僅啟用私有端點時，不會從網際網路公開存取您的 API 伺服器。前往您叢集 API 伺服器的所有流量必須來自叢集的 VPC 或連接的網路。節點會透過 VPC 中的 X-ENIs與 API 伺服器通訊。請注意，叢集管理工具必須能夠存取私有端點。進一步了解如何從 Amazon VPC 外部連線至私有 Amazon EKS 叢集端點。

請注意，公有 DNS 伺服器會將叢集的 API 伺服器端點解析為來自 VPC 的私有 IP 地址。在過去，端點只能從 VPC 內解析。

VPC 組態

Amazon VPC 支援 IPv4 和 IPv6 地址。Amazon EKS 預設支援 IPv4。VPC 必須具有與之建立關聯的 IPv4 CIDR 區塊。您可以選擇將多個 IPv4 無類別網域間路由 (CIDR) 區塊和多個 IPv6 CIDR 區塊與您的 VPC 建立關聯。建立 VPC 時，您必須從 RFC 1918 中指定的私有 IPv4 地址範圍指定 VPC 的 IPv4

CIDR 區塊。 http://www.faqs.org/rfcs/rfc1918.html允許區塊大小介於/16字首 (65，536 個 IP 地址）和/28字首 (16 個 IP 地址） 之間。

建立新的 VPC 時，您可以連接單一 IPv6 CIDR 區塊，變更現有 VPC 時最多可連接五個。IPv6 CIDR 區塊大小的字首長度可以介於 /44 和 /60 之間，而 IPv6 子網路的字首長度可以介於 /44/ 和 /64 之間。您可以從 Amazon 維護的 IPv6 地址集區請求 IPv6 CIDR 區塊。如需詳細資訊，請參閱 VPC 使用者指南中的 VPC CIDR 區塊一節。

Amazon EKS 叢集同時支援 IPv4 和 IPv6。根據預設，EKS 叢集會使用 IPv4 IP。在叢集建立時間指定 IPv6 將啟用使用 IPv6 叢集。IPv6 叢集需要雙堆疊 VPCs和子網路。

Amazon EKS 建議您在叢集建立期間，至少使用兩個位於不同可用區域的子網路。您在叢集建立期間傳入的子網路稱為叢集子網路。當您建立叢集時，Amazon EKS 會在您指定的子網路中建立最多 4 個跨帳戶 (x 帳戶或 x-ENIs) ENIs。x-ENIs 一律會部署，並用於叢集管理流量，例如日誌交付、EXEC 和 Proxy。如需完整的 VPC 和子網路需求詳細資訊，請參閱 EKS 使用者指南。

Kubernetes 工作者節點可以在叢集子網路中執行，但不建議這麼做。在叢集升級期間，Amazon EKS 會在叢集子網路中佈建其他 ENIs。當您的叢集向外擴展時，工作者節點和 Pod 可能會使用叢集子網路中的可用 IPs。因此，為了確保有足夠的可用 IPs，您可能會考慮將專用叢集子網路與 /28 網路遮罩搭配使用。

Kubernetes 工作者節點可以在公有或私有子網路中執行。子網路是公有還是私有，是指子網路內的流量是透過網際網路閘道路由。公有子網路具有透過網際網路閘道連至網際網路的路由表項目，但私有子網路則否。

源自於其他地方並到達節點的流量稱為傳入。源自節點且離開網路的流量稱為輸出。在以網際網路閘道設定的子網路中具有公有或彈性 IP 地址 EIPs) 的節點，允許從 VPC 外部傳入。私有子網路通常具有 NAT 閘道的路由，不允許從 VPC 外部將流量傳入子網路中的節點，同時仍然允許來自節點的流量離開 VPC (輸出)。

在 IPv6 世界中，每個地址都是網際網路路由。與節點和 Pod 相關聯的 IPv6 地址是公有的。在 VPC 中實作輸出限定網際網路閘道 (EIGW) 來支援私有子網路，允許傳出流量，同時封鎖所有傳入流量。您可以在 VPC 使用者指南中找到實作 IPv6 子網路的最佳實務。

您可以透過三種不同的方式設定 VPC 和子網路：

僅使用公有子網路

在相同的公有子網路中，會同時建立節點和輸入資源 （ 例如負載平衡器 ）。使用 標記公有子網路kubernetes.io/role/elb，以建構面向網際網路的負載平衡器。在此組態中，叢集端點可以設定為公有、私有或兩者 （ 公有和私有 ）。

使用私有和公有子網路

節點是在私有子網路上建立，而傳入資源會在公有子網路中執行個體化。您可以啟用對叢集端點的公有、私有或兩者（公有和私有）存取。根據叢集端點的組態，節點流量將透過 NAT 閘道或 ENI 進入。

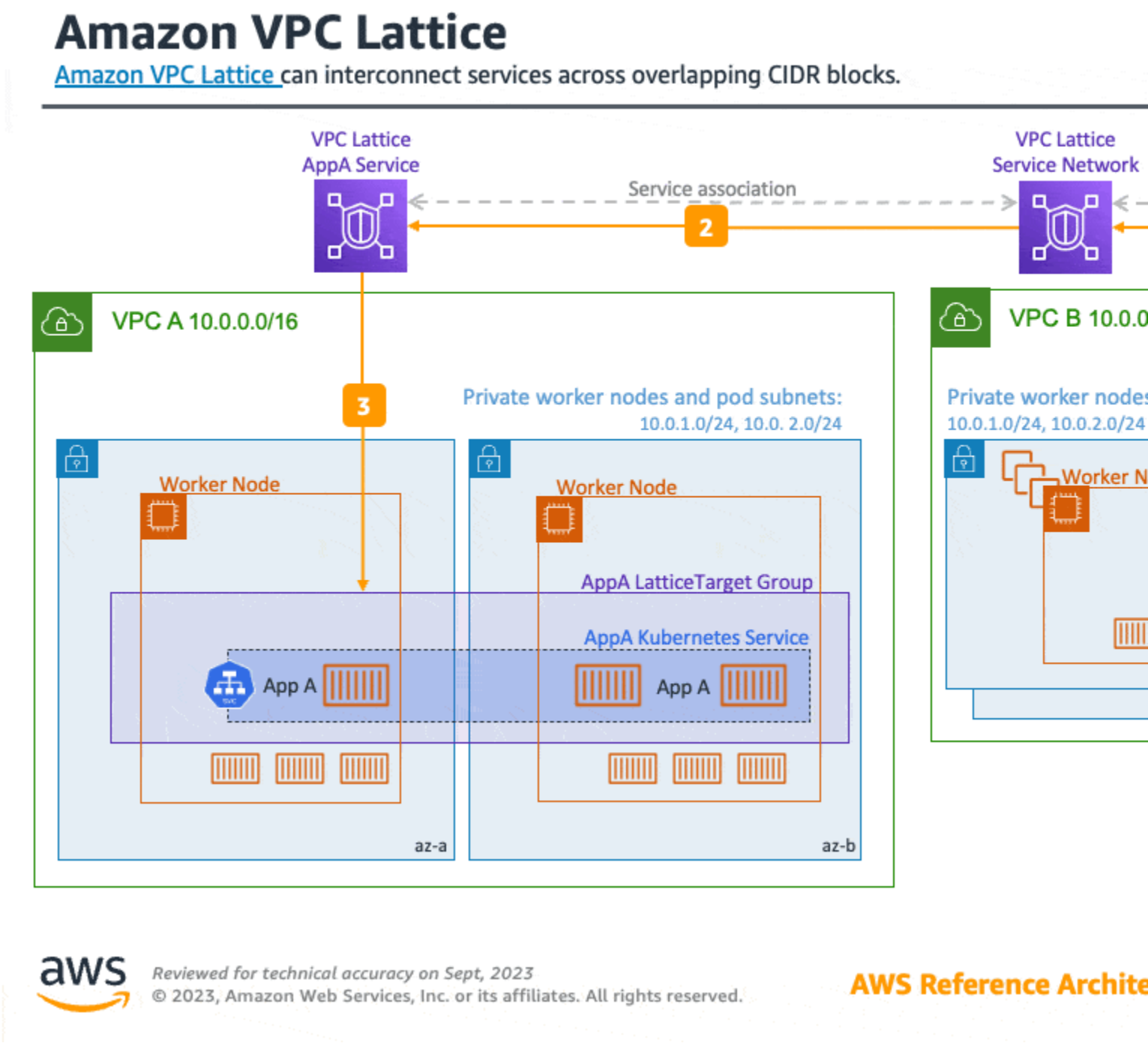
僅使用私有子網路

節點和輸入都會在私有子網路中建立。使用kubernetes.io/role/internal-elb子網路標籤來建構內部負載平衡器。存取叢集的端點需要 VPN 連線。您必須為 EC2 和所有 Amazon ECR 和 S3 儲存庫啟用 AWS PrivateLink。僅應啟用叢集的私有端點。建議您先完成 EKS 私有叢集需求，再佈建私有叢集。

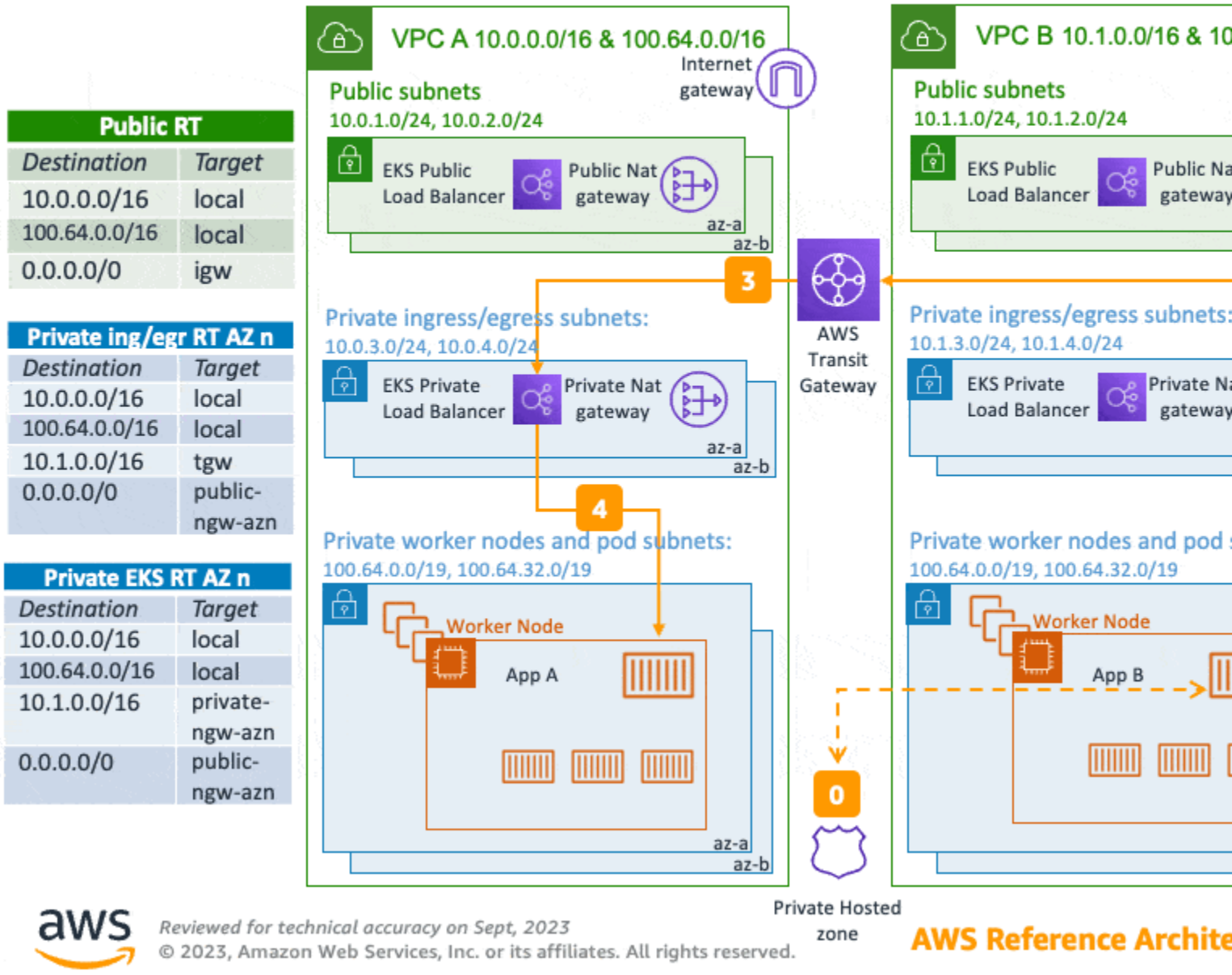
跨 VPCs通訊

在許多情況下，您需要部署到這些 VPCs多個 VPCs 和個別 EKS 叢集。

您可以使用 Amazon VPC Lattice 來一致且安全地跨多個 VPCs 和帳戶連接服務（無需由 VPC 對等互連、AWS PrivateLink 或 AWS Transit Gateway 等服務提供額外的連線）。在這裡進一步了解。



Amazon VPC Lattice 在 IPv4 和 IPv6 中的連結本機地址空間中運作，在可能有重疊 IPv4 地址的服務之間提供連線。為了提高營運效率，我們強烈建議將 EKS 叢集和節點部署到不重疊的 IP 範圍。如果您的基礎設施包含 IP 範圍重疊VPCs，您需要相應地建構您的網路。我們建議在自訂聯網模式下搭配傳輸閘道使用私有 NAT 閘道或 VPC CNI，以整合 EKS 上的工作負載，解決重複的 CIDR 挑戰，同時保留可路由 RFC1918 IP 地址。

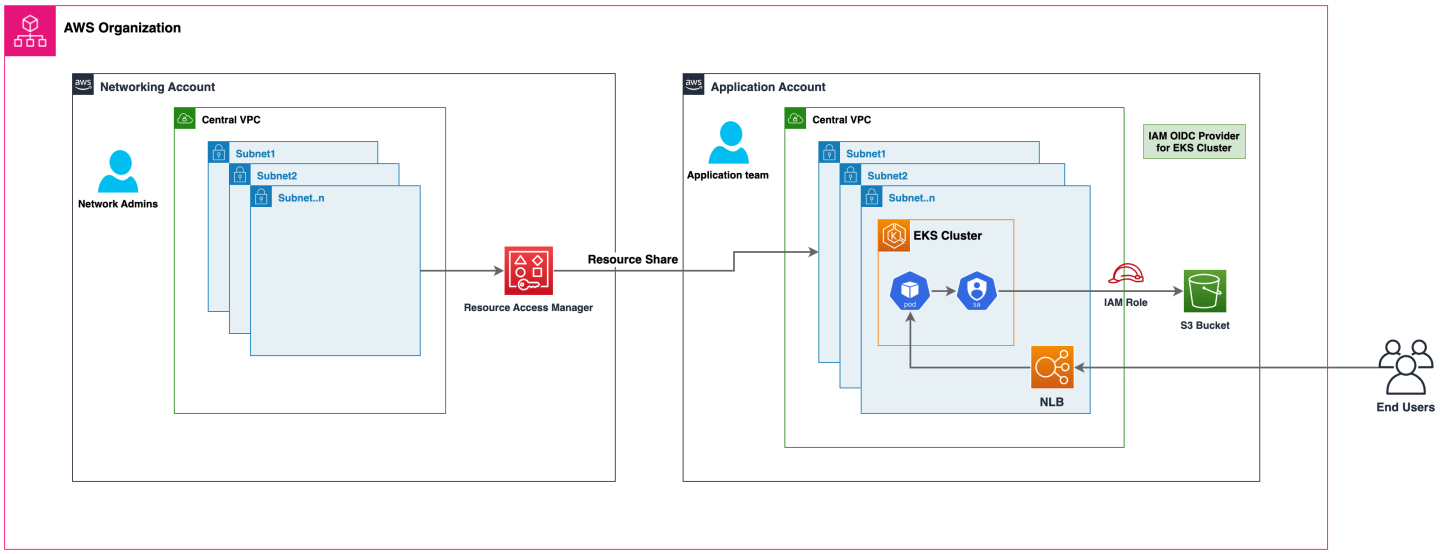


如果您是服務供應商，並且想要在不同的帳戶中與客戶 VPC 共用 Kubernetes 服務和輸入 (ALB 或 NLB)，請考慮使用 AWS PrivateLink，也稱為端點服務。

在多個帳戶之間共用 VPC

許多企業採用共用 Amazon VPCs 作為簡化網路管理、降低成本和改善 AWS 組織中多個 AWS 帳戶安全性的方法。他們利用 AWS Resource Access Manager (RAM) 安全地與個別 AWS 帳戶、組織單位 (OUs) 或整個 AWS Organization 共用支援的 AWS 資源。

您可以使用 AWS RAM，在另一個 AWS 帳戶的共用 VPC 子網路中部署 Amazon EKS 叢集、受管節點群組和其他支援 AWS 資源 （例如 LoadBalancers、安全群組、端點等）。下圖說明了範例高階架構。這可讓中央聯網團隊控制 VPCs、子網路等聯網建構，同時允許應用程式或平台團隊在各自的 AWS 帳戶中部署 Amazon EKS 叢集。此案例的完整演練可在此 github 儲存庫取得。



使用共用子網路時的考量事項

- Amazon EKS 叢集和工作者節點可以在屬於相同 VPC 的共用子網路內建立。Amazon EKS 不支援跨多個 VPCs建立叢集。
- Amazon EKS 使用 AWS VPC 安全群組 (SGs) 來控制 Kubernetes 控制平面與叢集工作者節點之間的流量。安全群組也用於控制工作者節點與其他 VPC 資源之間的流量，以及外部 IP 地址。您必須在應用程式/參與者帳戶中建立這些安全群組。確保您打算用於 Pod 的安全群組也位於參與者帳戶中。您可以在安全群組中設定傳入和傳出規則，以允許必要的流量往返中央 VPC 帳戶中的安全群組。
- 在 Amazon EKS 叢集所在的參與者帳戶中建立 IAM 角色和相關政策。這些 IAM 角色和政策對於授予必要許可給 Amazon EKS 管理的 Kubernetes 叢集，以及在 Fargate 上執行的節點和 Pod 至關重要。許可可讓 Amazon EKS 代表您呼叫其他 AWS 服務。
- 您可以遵循下列方法，允許跨帳戶從 k8s Pod 存取 AWS 資源，例如 Amazon S3 儲存貯體、Dynamodb 資料表等：

  - 資源型政策方法：如果 AWS 服務支援資源政策，您可以新增適當的資源型政策，以允許跨帳戶存取指派給 kubernetes Pod 的 IAM 角色。在此案例中，OIDC 提供者、IAM 角色和許可政策存在於應用程式帳戶中。若要尋找支援資源型政策的 AWS 服務，請參閱使用 IAM 的 AWS 服務，並在資源型欄中尋找具有是的服務。
  - OIDC 提供者方法：將在資源存在的其他參與者 AWS 帳戶中建立 IAM 資源，例如 OIDC 提供者、IAM 角色、許可和信任政策。這些角色將指派給應用程式帳戶中的 Kubernetes Pod，以便他們可以存取跨帳戶資源。如需此方法的完整演練，請參閱 Kubernetes 服務帳戶部落格的跨帳戶 IAM 角色。
- 您可以部署 Amazon Elastic Loadbalancer (ELB) 資源 (ALB 或 NLB)，將流量路由到應用程式或中央聯網帳戶中的 k8s Pod。如需在中央聯網帳戶中部署 ELB 資源的詳細說明，請參閱透過跨帳戶Load Balancer公開 Amazon EKS Pod。此選項提供增強的彈性，因為它會授予中央網路帳戶對 Load Balancer 資源安全組態的完整控制權。
- 使用 `custom networking feature` Amazon VPC CNI 時，您需要使用中央聯網帳戶中列出的可用區域 (AZ) ID 映射來建立每個 `ENIConfig`。這是因為實體 AZs 隨機映射到每個 AWS 帳戶中的 AZ 名稱。

安全群組

安全群組負責控制允許到達和離開其關聯資源的流量。Amazon EKS 使用安全群組來管理控制平面和節點之間的通訊。當您建立叢集時，Amazon EKS 會建立名稱為 `eks-cluster-sg-my-cluster-uniqueID` 的安全群組。EKS 將這些安全群組與受管 ENIs和節點建立關聯。此安全群組的設計目的是允許來自控制平面和 受管節點群組的所有流量彼此之間可以自由流動，以及允許所有的輸出流量流往任何目的地。

建立叢集時，您可以指定自己的安全群組。當您指定自己的安全群組時，請參閱安全群組的建議。

建議

考慮異地同步備份部署

AWS 區域提供多個實體隔離和隔離的可用區域 (AZ)，這些可用區域與低延遲、高輸送量和高備援聯網連接。透過可用區域，您可以設計和操作在可用區域之間自動容錯移轉的應用程式，而不會中斷。Amazon EKS 強烈建議將 EKS 叢集部署到多個可用區域。當您建立叢集時，請考慮在至少兩個可用區域中指定子網路。

在節點上執行的 Kubelet 會自動將標籤新增至節點物件，例如 `topology.kubernetes.io/region=us-west-2`。我們建議您使用節點標籤搭配 Pod 拓撲分散限制，以控制 Pod 如何分散到各

個區域。這些提示可讓 Kubernetes 排程器放置 Pod 以獲得更好的預期可用性，從而降低相關故障影響整個工作負載的風險。請參閱將節點指派給 Pod，以查看節點選擇器和可用區域分散限制的範例。

您可以在建立節點時定義子網路或可用區域。如果未設定子網路，節點會放置在叢集子網路中。受管節點群組的 EKS 支援會自動將節點分散到可用容量的多個可用區域。如果工作負載定義拓撲分散限制，Karpenter 將透過將節點擴展到指定的 AZs 來遵守 AZ 分散置放。

AWS Elastic Load Balancer 由 Kubernetes 叢集的 AWS Load Balancer 控制器管理。它為 Kubernetes 輸入資源佈建 Application Load Balancer (ALB)，並為 Loadbalancer 類型的 Kubernetes 服務佈建 Network Load Balancer (NLB)。Elastic Load Balancer 控制器使用標籤來探索子網路。ELB 控制器至少需要兩個可用區域 (AZs)，才能成功佈建輸入資源。請考慮在至少兩個 AZs 中設定子網路，以利用地理備援的安全性和可靠性。

將節點部署至私有子網路

包含私有和公有子網路的 VPC 是在 EKS 上部署 Kubernetes 工作負載的理想方法。請考慮在兩個不同的可用區域中設定至少兩個公有子網路和兩個私有子網路。公有子網路的相關路由表包含網際網路閘道的路由。Pod 可以透過 NAT 閘道與網際網路互動。IPv6 環境 (EIGW) 中的輸出限定網際網路閘道支援私有子網路。

執行個體化私有子網路中的節點可對節點的流量提供最大控制，並對絕大多數的 Kubernetes 應用程式有效。傳入資源（例如負載平衡器）會在公有子網路中執行個體化，並將流量路由到在私有子網路上操作的 Pod。

如果您需要嚴格的安全和網路隔離，請考慮僅限私有模式。在此組態中，三個私有子網路會部署在 AWS 區域的 VPC 內不同的可用區域中。部署到子網路的資源無法存取網際網路，網際網路也無法存取子網路中的資源。若要讓您的 Kubernetes 應用程式存取其他 AWS 服務，您必須設定 PrivateLink 介面和/或閘道端點。您可以使用 AWS Load Balancer 控制器設定內部負載平衡器，將流量重新導向至 Pod。私有子網路必須加上標籤 (`kubernetes.io/role/internal-elb: 1`)，控制器才能佈建負載平衡器。若要讓節點向叢集註冊，叢集端點必須設定為私有模式。如需完整的需求和考量事項，請參閱私有叢集指南。

考慮叢集端點的公有和私有模式

Amazon EKS 提供僅限公有、public-and-private和僅限私有的叢集端點模式。預設模式僅限公有，但我們建議在公有和私有模式中設定叢集端點。此選項允許叢集 VPC 內的 Kubernetes API 呼叫（例如 node-to-control-plane通訊）利用私有 VPC 端點和流量來保留在叢集的 VPC 內。另一方面，您可以從網際網路連線您的叢集 API 伺服器。不過，我們強烈建議限制可以使用公有端點的 CIDR 區塊。了解如何設定公有和私有端點存取，包括限制 CIDR 區塊。

當您需要安全和網路隔離時，我們建議使用僅限私有的端點。我們建議您使用 EKS 使用者指南中列出的任一選項，私下連線至 API 伺服器。

仔細設定安全群組

Amazon EKS 支援使用自訂安全群組。任何自訂安全群組都必須允許節點與 Kubernetes 控制平面之間的通訊。當您的組織不允許開放式通訊時，請檢查連接埠需求並手動設定規則。

EKS 會將您在叢集建立期間提供的自訂安全群組套用至受管介面 (X-ENIs)。不過，它不會立即將它們與節點建立關聯。建立節點群組時，強烈建議手動關聯自訂安全群組。請考慮啟用 securityGroupSelectorTerms，以在節點自動擴展期間啟用自訂安全群組的 Karpenter 節點範本探索。

我們強烈建議建立安全群組，以允許所有節點間通訊流量。在引導程序期間，節點需要傳出網際網路連線才能存取叢集端點。評估外部存取需求，例如內部部署連線和容器登錄檔存取，並適當設定規則。在將變更放入生產環境之前，我們強烈建議您在開發環境中仔細檢查連線。

在每個可用區域中部署 NAT 閘道

如果您在私有子網路 (IPv4 和 IPv6) 中部署節點，請考慮在每個可用區域 (AZ) 中建立 NAT 閘道，以確保區域獨立架構並減少跨可用區域支出。AZ 中的每個 NAT 閘道都會以備援實作。

Amazon VPC CNI

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

Amazon EKS 透過 Amazon VPC 容器網路介面外掛程式實作叢集聯網，也稱為 VPC CNI。CNI 外掛程式可讓 Kubernetes Pod 擁有與 VPC 網路相同的 IP 地址。更具體地說，Pod 內的所有容器都會共用網路命名空間，而且可以使用本機連接埠彼此通訊。

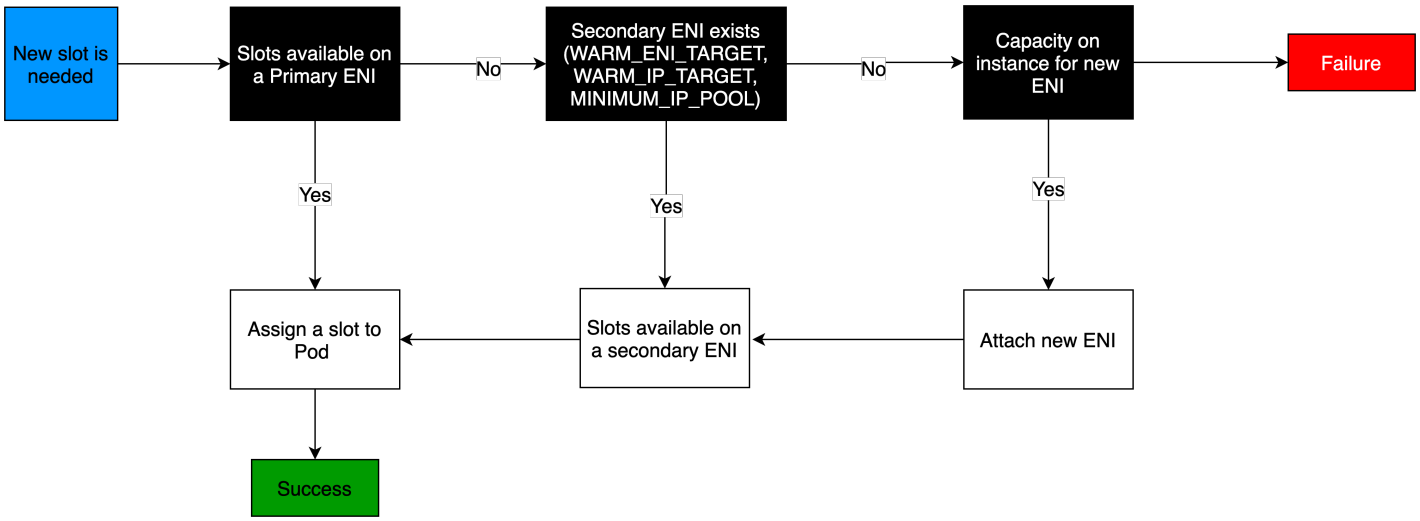
Amazon VPC CNI 有兩個元件：

- CNI Binary，這會設定 Pod 網路以啟用 Pod-to-Pod 通訊。CNI 二進位檔案會在節點根檔案系統上執行，並在新增 Pod 或從節點移除現有 Pod 時，由 kubelet 叫用。
- ipamd，長時間執行的節點本機 IP 地址管理 (IPAM) 協助程式，負責：
  - 管理節點上的 ENIs，以及
  - 維護可用 IP 地址或字首的暖集區

建立執行個體時，EC2 會建立並連接與主要子網路相關聯的主要 ENI。主要子網路可以是公有或私有。在 hostNetwork 模式下執行的 Pod 會使用指派給節點主要 ENI 的主要 IP 地址，並共用與主機相同的網路命名空間。

CNI 外掛程式會管理節點上的彈性網路界面 (ENI)。佈建節點時，CNI 外掛程式會自動將插槽集區 (IPs 或字首） 從節點的子網路配置到主要 ENI。此集區稱為暖集區，其大小取決於節點的執行個體類型。根據 CNI 設定，插槽可以是 IP 地址或字首。指派 ENI 上的插槽後，CNI 可能會將具有暖槽集區的其他 ENIs 連接到節點。這些額外的 ENIs稱為次要 ENIs。根據執行個體類型，每個 ENI 只能支援特定數量

的槽。CNI 會根據所需的插槽數量，將更多 ENIs 連接至執行個體，這通常對應至 Pod 的數量。此程序會持續進行，直到節點不再支援其他 ENI 為止。CNI 也會預先配置「暖」ENIs和插槽，以加快 Pod 啟動速度。請注意，每個執行個體類型都有可連接的 ENIs 數量上限。除了運算資源之外，這是 Pod 密度（每個節點的 Pod 數量）的一個限制。



網路介面的數量上限，以及您可以使用的插槽數量上限，會因 EC2 執行個體的類型而有所不同。由於每個 Pod 會在插槽上使用 IP 地址，因此您可以在特定 EC2 執行個體上執行的 Pod 數量取決於可以連接多少個 ENIs，以及每個 ENI 支援的插槽數量。我們建議為每個 EKS 使用者指南設定最大 Pod，以避免耗盡執行個體的 CPU 和記憶體資源。使用此計算`hostNetwork`排除使用 的 Pod。如需詳細資訊，請參閱《Amazon EKS 使用者指南》中的 maxPods的判斷方式。

概觀

次要 IP 模式是 VPC CNI 的預設模式。本指南提供啟用次要 IP 模式時 VPC CNI 行為的一般概觀。ipamd 的功能 (IP 地址的配置）可能會因 VPC CNI 的組態設定而有所不同，例如 the section called “Linux 的字首模式”、 the section called “每個 Pod 的安全群組”和 the section called “自訂聯網”。

Amazon VPC CNI 部署為工作者節點上名為 aws-node 的 Kubernetes Daemonset。佈建工作者節點時，會附加一個預設 ENI，稱為主要 ENI。CNI 會從連接至節點主要 ENIs 的子網路配置 ENI 和次要 IP 地址的暖集區。根據預設，ipamd 會嘗試將額外的 ENI 配置給節點。IPAMD 會在排程單一 Pod 時配置額外的 ENI，並從主要 ENI 指派次要 IP 地址。此「暖」ENI 可讓 Pod 聯網速度更快。當次要 IP 地址集區用完時，CNI 會新增另一個 ENI 來指派更多地址。

集區中的 ENIs 和 IP 地址數量是透過 WARM_ENI_TARGET、WARM_IP_TARGET、MINIMUM_IP_TARGET 等環境變數來設定。`aws-`

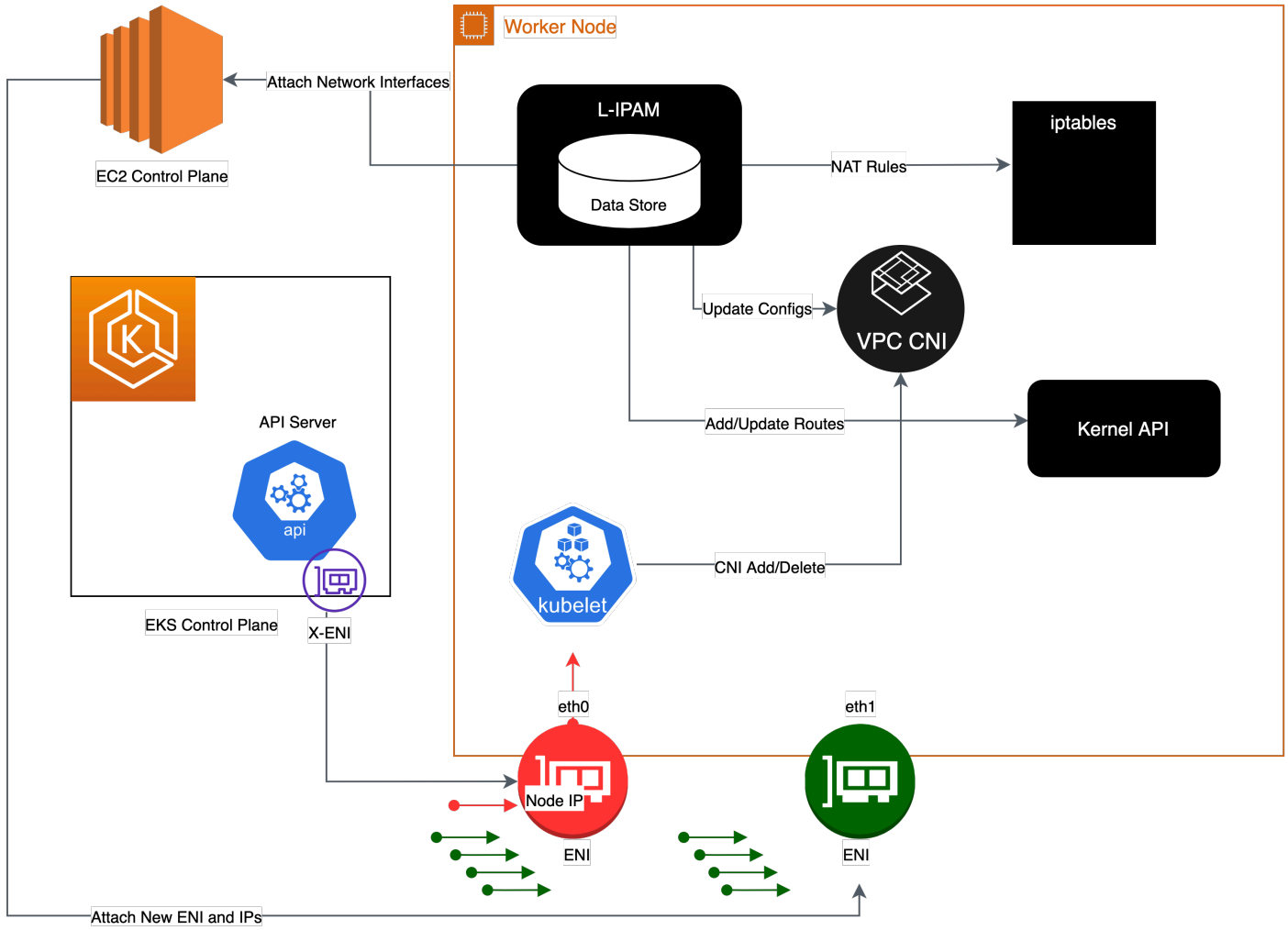
node Daemonset 會定期檢查是否已連接足夠數量ENIs。當符合所有 WARM_ENI_TARGET、 或 WARM_IP_TARGET和 MINIMUM_IP_TARGET條件時，會連接足夠數量ENIs。如果連接的 ENIs不足，CNI 會呼叫 EC2 以連接更多 ENI，直到達到MAX_ENI限制為止。

- WARM_ENI_TARGET - 整數，大於 0 的值表示要求已啟用
  - 要維護ENIs 數目。ENI 作為次要 ENI 連接到節點時為「暖」，但任何 Pod 都不會使用它。更具體地說，沒有 ENI 的 IP 地址與 Pod 相關聯。
  - 範例：考慮具有 2 個 ENIs執行個體，每個 ENI 都支援 5 個 IP 地址。WARM_ENI_TARGET 設定為 1。如果只有 5 個 IP 地址與執行個體相關聯，CNI 會維護連接到執行個體的 2 ENIs。第一個 ENI 正在使用中，並使用此 ENI 的所有 5 個可能 IP 地址。第二個 ENI 是「暖」，集區中全部有 5 個 IP 地址。如果在執行個體上啟動另一個 Pod，則需要第 6 個 IP 地址。CNI 會將來自第二個 ENI 和來自集區 5 個 IPs 的 IP 地址指派給此第 6 個 Pod。第二個 ENI 現在正在使用中，不再處於「暖」狀態。CNI 將配置第 3 個 ENI，以維持至少 1 個暖 ENI。

> **Note**
>
> 暖 ENIs 仍會使用來自 VPC CIDR 的 IP 地址。IP 地址「未使用」或「暖」，直到它們與工作負載相關聯，例如 Pod。

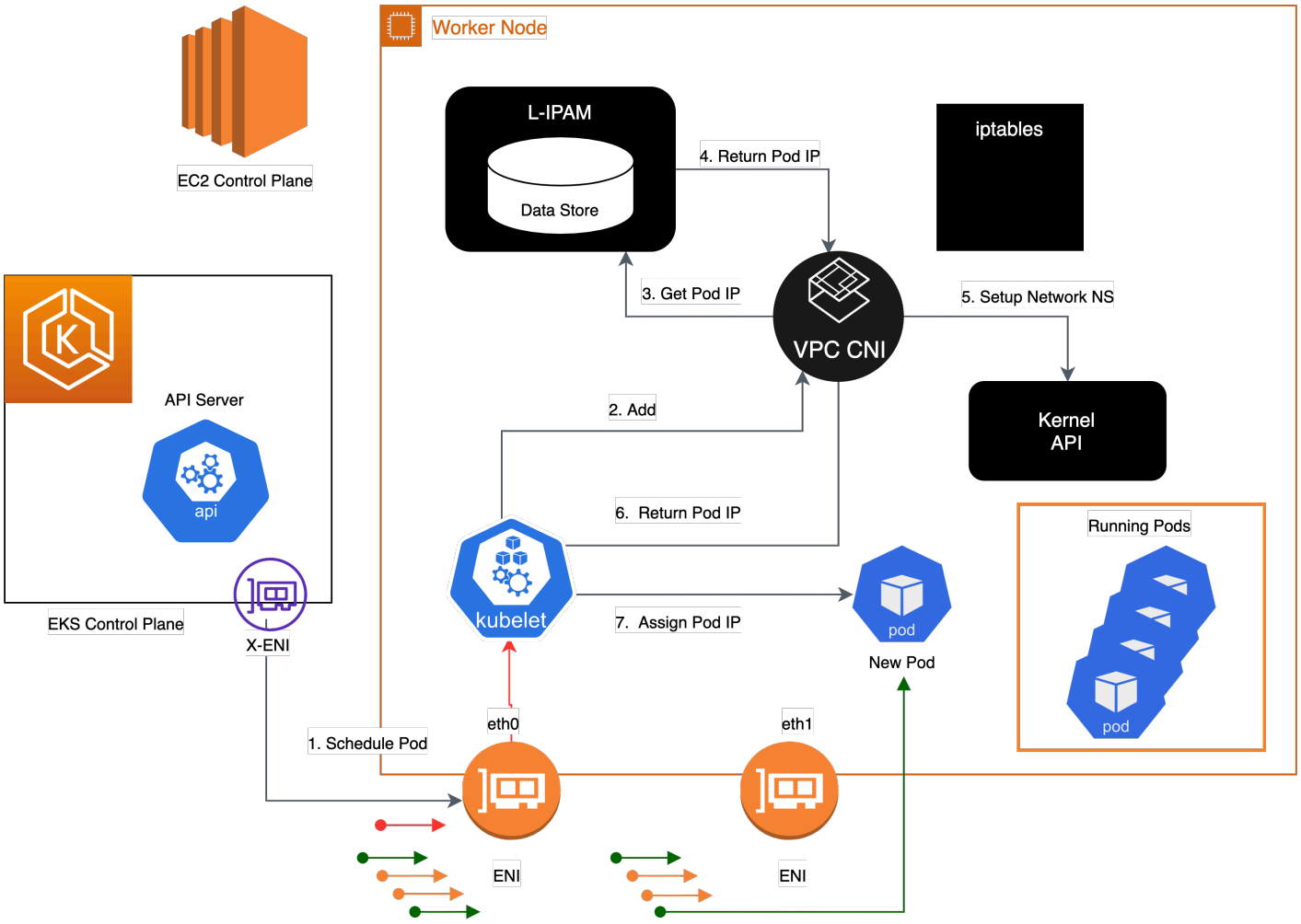
- WARM_IP_TARGET、整數、大於 0 的值表示要求已啟用
  - 要維護的暖 IP 地址數目。暖 IP 可在主動連接的 ENI 上使用，但尚未指派給 Pod。換句話說，可用的暖 IPs 數量是可指派給 Pod 的 IPs 數量，而不需要額外的 ENI。
- 範例：考慮具有 1 個 ENI 的執行個體，每個 ENI 支援 20 個 IP 地址。WARM_IP_TARGET 設定為 5。WARM_ENI_TARGET 設定為 0。在需要第 16 個 IP 地址之前，只會連接 1 個 ENI。然後，CNI 會連接第二個 ENI，從子網路 CIDR 耗用 20 個可能的地址。
- MINIMUM_IP_TARGET、整數、大於 0 的值表示要求已啟用
  - 隨時要配置的 IP 地址數量下限。這通常用於在執行個體啟動時預先載入多個 ENIs的指派。
  - 範例：考慮新啟動的執行個體。它有 1 個 ENI，每個 ENI 支援 10 個 IP 地址。MINIMUM_IP_TARGET 設定為 100。ENI 會立即連接 9 個 ENIs，總共連接 100 個地址。無論任何 WARM_IP_TARGET 或 WARM_ENI_TARGET 值為何，都會發生這種情況。

此專案包含子網路計算器 Excel 文件。此計算器文件會模擬不同 ENI 組態選項下指定工作負載的 IP 地址使用量，例如 WARM_IP_TARGET和 WARM_ENI_TARGET。

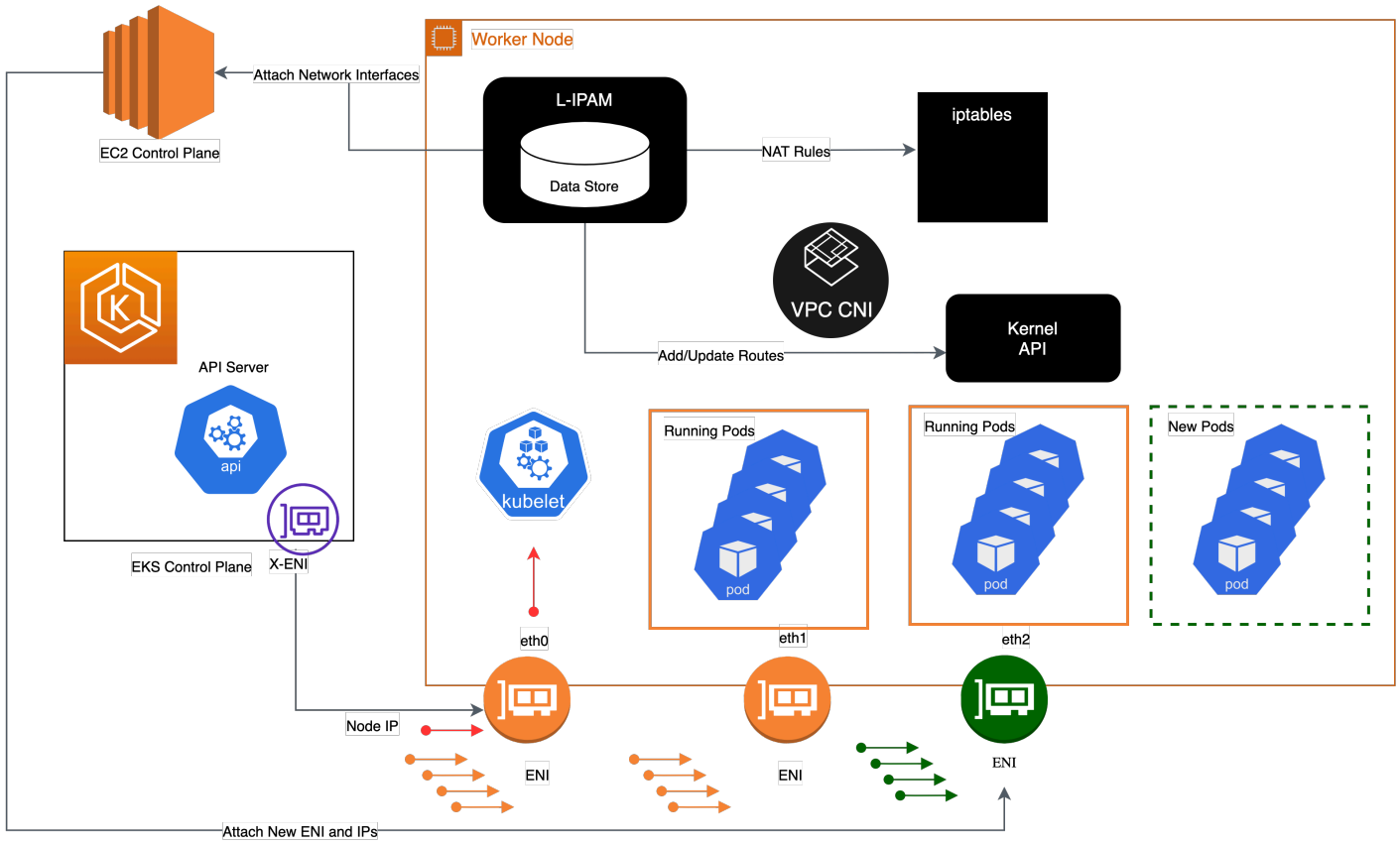


當 Kubelet 收到新增 Pod 請求時，CNI 二進位查詢會偵測可用的 IP 地址，然後 ipamd 會提供給 Pod。CNI 二進位檔會連接主機和 Pod 網路。

根據預設，部署在節點上的 Pod 會指派給與主要 ENI 相同的安全群組。或者，Pod 可以設定不同的安全群組。



當 IP 地址集區耗盡時，外掛程式會自動將另一個彈性網路介面連接至執行個體，並將另一組次要 IP 地址分配給該介面。此程序會持續進行，直到節點不再支援其他彈性網路介面為止。



刪除 Pod 時，VPC CNI 會將 Pod 的 IP 地址放在 30 秒的冷卻快取中。冷卻快取中的 IPs 不會指派給新的 Pod。當冷靜期結束時，VPC CNI 會將 Pod IP 移回暖集區。冷靜期可防止 Pod IP 地址過早回收，並允許所有叢集節點上的 kube-proxy 完成更新 iptables 規則。當 IPs 或 ENIs 的數量超過暖集區設定的數量時，ipamd 外掛程式會將 IPs 和 ENIs 傳回 VPC。

如上所述，在次要 IP 模式中，每個 Pod 會從連接至執行個體的其中一個 ENIs 接收一個次要私有 IP 地址。由於每個 Pod 使用 IP 地址，因此您可以在特定 EC2 執行個體上執行的 Pod 數量取決於可以連接多少個 ENIs 及其支援的 IP 地址。VPC CNI 會檢查限制檔案，以了解每種執行個體類型允許多少 ENIs和 IP 地址。

您可以使用下列公式來判斷您可以在節點上部署的 Pod 數量上限。

```
(Number of network interfaces for the instance type * (the number of IP addresses per
 network interface - 1)) + 2
```

+2 表示需要主機聯網的 Pod，例如 kube-proxy 和 VPC CNI。Amazon EKS 要求 kube-proxy 和 VPC CNI 在每個節點上操作，這些要求會納入最大 Pod 值。如果您想要執行其他主機聯網 Pod，請考慮更新最大 Pod 值。您可以在啟動範本中將 指定`--kubelet-extra-args "–max-pods=110"`為使用者資料。

例如，在具有 3 個 c5.large 節點（每個 ENIs 3 個 ENI 和最多 10 IPs) 的叢集上，當叢集啟動且具有 2 個 CoreDNS Pod 時，CNI 將使用 50 個 IP 地址，並將 43 個 IPs 保留在暖集區中。部署應用程式時，暖集區可讓 Pod 啟動更快速。

節點 1（含 CoreDNS Pod)：2 個 ENIs，20 個已指派 IPs

節點 2（使用 CoreDNS Pod)：2 個 ENIs、20 個指派IPs

節點 3（無 Pod)：1 ENI。已指派 10 IPs。

對於節點 1 和節點 2（相同組態）：

- 每個 ENIs 2 個 ENI × 10 IPs = 總計 20 IPs
- 減去 2 IPs（每個 ENI 1 個）= 18 IPs
- 減去 1 個適用於 CoreDNS Pod 的 IP = 17 IPs
- 因此，每個節點在暖集區中都有 17 IPs

對於節點 3：

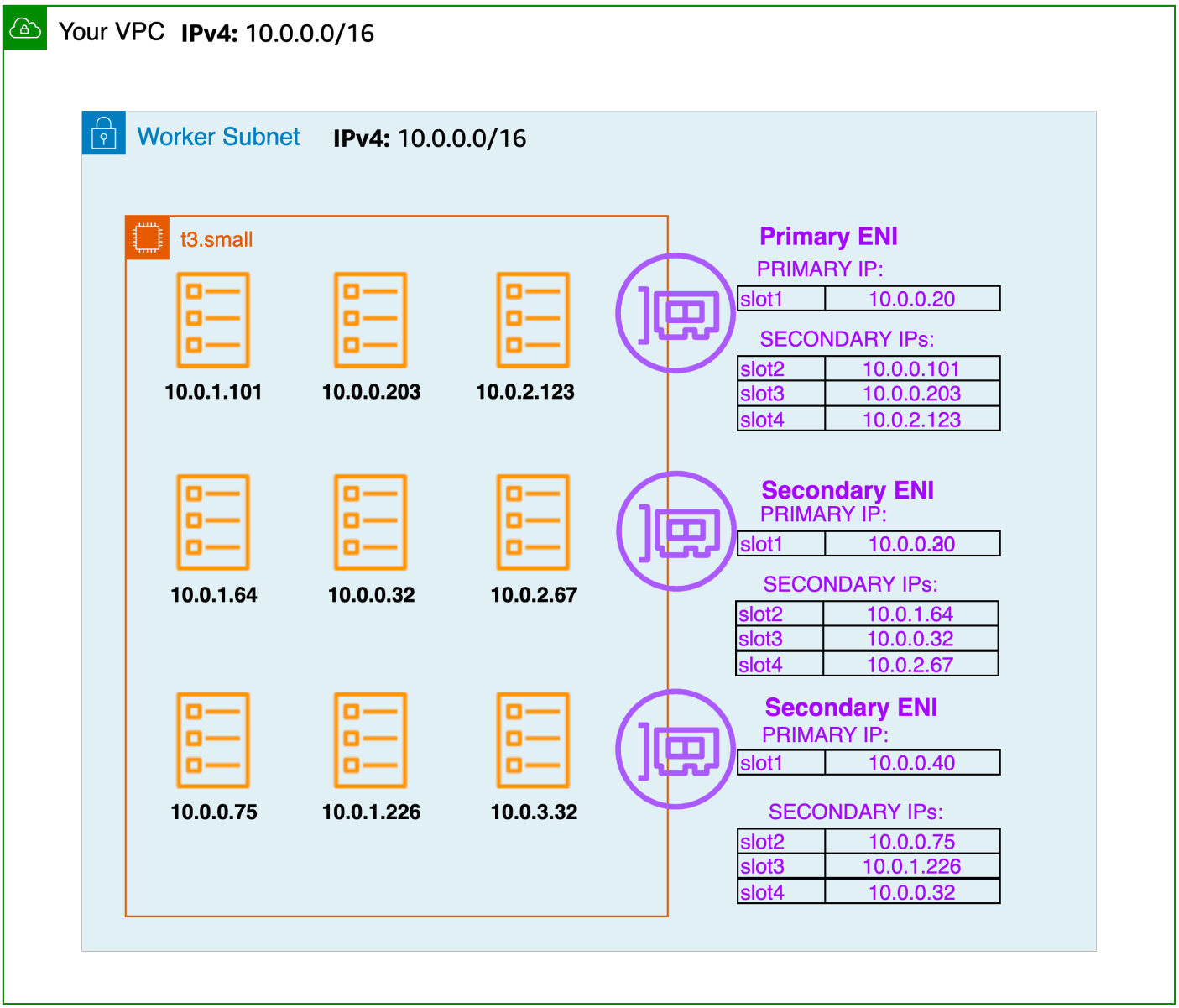
- 1 個 ENI × 10 IPs = 總計 10 IPs
- 減去 1 個主要 IP = 暖集區中可用的 9 IPs

總暖集區計算：- 17（節點 1) + 17（節點 2) + 9（節點 3) = 43 IPs

請記住，基礎設施 Pod，通常以協助程式集的形式執行，每個 Pod 都有助於達到最大 Pod 計數。這些可能包括：

- CoreDNS
- Amazon Elastic LoadBalancer
- 指標伺服器的操作 Pod

我們建議您結合這些 Pod 的容量來規劃基礎設施。如需每個執行個體類型支援的 Pod 數量上限清單，請參閱 GitHub 上的 eni-max-pods.txt。



建議

使用自動模式部署 EKS 叢集

當您使用 EKS Auto 模式建立叢集時，AWS 會管理叢集的 VPC 容器網路界面 (CNI) 組態。使用 Amazon EKS 自動模式，您無需安裝或升級聯網附加元件。不過，請確定您的工作負載與受管 VPC CNI 組態相容。

部署 VPC CNI 受管附加元件

當您佈建叢集時，Amazon EKS 會自動安裝 VPC CNI。不過，Amazon EKS 支援受管附加元件，可讓叢集與運算、儲存和聯網等基礎 AWS 資源互動。強烈建議您部署具有受管附加元件的叢集，包括 VPC CNI。

Amazon EKS 受管附加元件提供 Amazon EKS 叢集的 VPC CNI 安裝和管理。Amazon EKS 附加元件包含最新的安全修補程式、錯誤修正，並由 AWS 驗證，以使用 Amazon EKS。VPC CNI 附加元件可讓您持續確保 Amazon EKS 叢集的安全性和穩定性，並減少安裝、設定和更新附加元件所需的工作量。此外，可透過 Amazon EKS API、AWS 管理主控台、AWS CLI 和 eksctl 來新增、更新或刪除受管附加元件。

您可以使用 `--show-managed-fields` 旗標搭配 `kubectl get`命令來尋找 VPC CNI 的受管欄位。

```
kubectl get daemonset aws-node --show-managed-fields -n kube-system -o yaml
```

受管附加元件每 15 分鐘會自動覆寫組態，以防止組態偏離。這表示在建立附加元件後透過 Kubernetes API 對受管附加元件所做的任何變更，都會被自動偏離預防程序覆寫，並在附加元件更新程序期間設定為預設值。

EKS 管理的欄位會列在 managedFields 下，管理員為 EKS。EKS 管理的欄位包括服務帳戶、映像、映像 URL、活體探查、整備探查、標籤、磁碟區和磁碟區掛載。

> **Note**
>
> WARM_ENI_TARGET、WARM_IP_TARGET 和 MINIMUM_IP_TARGET 等最常用欄位不會受管，也不會進行調校。這些欄位的變更會在更新附加元件時保留。

我們建議您在更新生產叢集之前，先測試非生產叢集中特定組態的附加元件行為。此外，請遵循 EKS 使用者指南中的附加元件組態步驟。

遷移至受管附加元件

您將管理版本相容性，並更新自我管理 VPC CNI 的安全修補程式。若要更新自我管理的附加元件，您必須使用 EKS 使用者指南中概述的 Kubernetes APIs 和說明。我們建議遷移至現有 EKS 叢集的受管附加元件，並強烈建議在遷移之前建立目前 CNI 設定的備份。若要設定受管附加元件，您可以使用 Amazon EKS API、AWS 管理主控台或 AWS 命令列界面。

```
kubectl apply view-last-applied daemonset aws-node -n kube-system  aws-k8s-cni-old.yaml
```

如果 欄位列為使用預設設定管理，Amazon EKS 將取代 CNI 組態設定。我們注意不要修改受管欄位。附加元件不會協調暖環境變數和 CNI 模式等組態欄位。當您遷移至受管 CNI 時，Pod 和應用程式將繼續執行。

更新前備份 CNI 設定

VPC CNI 會在客戶資料平面 （節點） 上執行，因此 Amazon EKS 不會在發行新版本時或將叢集更新為新的 Kubernetes 次要版本後自動更新附加元件 （受管和自我管理）。若要更新現有叢集的附加元件，您必須透過 update-addon API 觸發更新，或按一下 EKS 主控台中附加元件的立即更新連結。如果您已部署自我管理附加元件，請遵循更新自我管理 VPC CNI 附加元件所述的步驟。

我們強烈建議您一次更新一個次要版本。例如，如果目前的次要版本是 `1.9`，並且想要更新為 `1.11`，您應先更新至 `1.10` 的最新修補程式版本，然後更新至 `1.11` 的最新修補程式版本。

在更新 Amazon VPC CNI 之前，執行 aws-node Daemonset 的檢查。備份現有設定。如果使用受管附加元件，請確認您尚未更新 Amazon EKS 可能覆寫的任何設定。我們建議您在自動化工作流程中使用更新後掛鉤，或在附加元件更新後手動套用步驟。

```
kubectl apply view-last-applied daemonset aws-node -n kube-system  aws-k8s-cni-old.yaml
```

對於自我管理的附加元件，請將備份與 GitHub `releases`上的 進行比較，以查看可用的版本，並熟悉您要更新之版本中的變更。建議使用 Helm 來管理自我管理的附加元件，並利用值檔案來套用設定。任何涉及 Daemonset 刪除的更新操作都會導致應用程式停機，且必須避免。

了解安全內容

我們強烈建議您了解為有效管理 VPC CNI 而設定的安全內容。Amazon VPC CNI 有兩個元件 CNI 二進位和 ipamd (aws-node) Daemonset。CNI 在節點上以二進位形式執行，並可存取節點根檔案系統，在節點層級處理 iptable 時也具有特殊存取權限。新增或移除 Pod 時，kubelet 會叫用 CNI 二進位檔。

aws-node Daemonset 是長期執行的程序，負責節點層級的 IP 地址管理。aws-node 會在 `hostNetwork` 模式下執行，並允許存取迴圈裝置，以及相同節點上其他 Pod 的網路活動。aws-node init 容器會以特殊權限模式執行，並掛載 CRI 通訊端，允許 Daemonset 監控節點上執行之 Pod 的 IP 用量。Amazon EKS 正在努力移除 aws-node init 容器的特權要求。此外，aws-node 需要更新 NAT 項目和載入 iptables 模組，因此使用 NET_ADMIN 權限執行。

Amazon EKS 建議部署 aws-node 資訊清單所定義的安全政策，以進行 Pod 和網路設定的 IP 管理。請考慮更新至最新版本的 VPC CNI。此外，如果您有特定的安全需求，請考慮開啟 GitHub 問題。

針對 CNI 使用個別的 IAM 角色

AWS VPC CNI 需要 AWS Identity and Access Management (IAM) 許可。必須先設定 CNI 政策，才能使用 IAM 角色。您可以使用 AmazonEKS_CNI_Policy，這是 IPv4 叢集的 AWS 受管政策。AmazonEKS CNI 受管政策只有 IPv4 叢集的許可。您必須為具有此處所列許可的 IPv6 叢集建立單獨的 IAM 政策。

根據預設，VPC CNI 會繼承 Amazon EKS 節點 IAM 角色（受管節點和自我管理節點群組）。

強烈建議使用 Amazon VPC CNI 的相關政策設定個別 IAM 角色。如果沒有，Amazon VPC CNI 的 Pod 會取得指派給節點 IAM 角色的許可，並可存取指派給節點的執行個體描述檔。

VPC CNI 外掛程式會建立和設定稱為 aws-node 的服務帳戶。根據預設，服務帳戶會繫結至已連接 Amazon EKS CNI 政策的 Amazon EKS 節點 IAM 角色。若要使用個別的 IAM 角色，建議您建立連接 Amazon EKS CNI 政策的新服務帳戶。若要使用新的服務帳戶，您必須重新部署 CNI Pod。建立新叢集時，請考慮`--service-account-role-arn`為 VPC CNI 受管附加元件指定 。請務必從 Amazon EKS 節點角色移除 IPv4 和 IPv6 的 Amazon EKS CNI 政策。

建議您封鎖存取執行個體中繼資料，以將安全性漏洞的爆量半徑降至最低。

處理活體/就緒探查失敗

我們建議提高 EKS 1.20 和更新版本叢集的存留期和整備探查逾時值（預設 `timeoutSeconds: 10`），以防止探查失敗導致應用程式的 Pod 卡在 containerCreating 狀態。此問題已在資料密集型和批次處理叢集中出現。高 CPU 使用量會導致 aws 節點探查運作狀態失敗，導致 Pod CPU 請求未滿足。除了修改探查逾時之外，請確定已正確設定 aws-node 的 CPU 資源請求（預設 `CPU: 25m`）。除非您的節點發生問題，否則我們不建議更新設定。

當您使用 Amazon EKS 支援時，我們強烈建議您在節點`bash /opt/cni/bin/aws-cni-support.sh`上執行 sudo。指令碼將協助評估節點上的 kubelet 日誌和記憶體使用率。請考慮在 Amazon EKS 工作者節點上安裝 SSM Agent 來執行指令碼。

在非 EKS 最佳化 AMI 執行個體上設定 IPTables 轉送政策

如果您使用的是自訂 AMI，請務必將 iptable 轉送政策設定為 kubelet.service 下的 ACCEPT。許多系統會將 iptable 轉送政策設定為 DROP。您可以使用 HashiCorp Packer 建置自訂 AMI，並使用 AWS GitHub 上 Amazon EKS AMI 儲存庫中的資源和組態指令碼建置規格。您可以更新 kubelet.service，並遵循此處指定的指示來建立自訂 AMI。

定期升級 CNI 版本

VPC CNI 可回溯相容。最新版本適用於所有 Amazon EKS 支援的 Kubernetes 版本。此外，VPC CNI 以 EKS 附加元件的形式提供（請參閱上述「部署 VPC CNI 受管附加元件」）。雖然 EKS 附加元件會協調附加元件的升級，但它不會像 CNI 一樣自動升級附加元件，因為它們在資料平面上執行。您有責任在受管和自我管理的工作者節點升級之後升級 VPC CNI 附加元件。

最佳化 IP 地址使用率

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

由於應用程式現代化，容器化環境正在快速擴展。這表示正在部署越來越多的工作者節點和 Pod。

Amazon VPC CNI 外掛程式會從 VPC 的 CIDR (s) 指派每個 Pod 的 IP 地址。此方法透過 VPC 流程日誌和其他監控解決方案等工具，提供 Pod 地址的完整可見性。根據您的工作負載類型，這可能會導致 Pod 耗用大量 IP 地址。

設計 AWS 網路架構時，請務必在 VPC 和節點層級最佳化 Amazon EKS IP 使用量。這可協助您緩解 IP 耗盡問題，並提高每個節點的 Pod 密度。

在本節中，我們將討論可協助您實現這些目標的技巧。

最佳化節點層級 IP 消耗

字首委派是 Amazon Virtual Private Cloud (Amazon VPC) 的一項功能，可讓您將 IPv4 或 IPv6 字首指派給 Amazon Elastic Compute Cloud (Amazon EC2) 執行個體。它會增加每個網路界面 (ENI) 的 IP 地址，這會提高每個節點的 Pod 密度並改善您的運算效率。自訂聯網也支援字首委派。

如需詳細資訊，請參閱 Linux 節點的字首委派和 Windows 節點的字首委派一節。

減少 IP 耗盡

為了防止叢集耗用所有可用的 IP 地址，我們強烈建議調整 VPCs 和子網路的大小，並考量其成長。

採用 IPv6 是從一開始就避免這些問題的好方法。不過，對於可擴展性需求超過初始規劃且無法採用 IPv6 的組織，改善 VPC 設計是對 IP 地址耗盡的建議回應。Amazon EKS 客戶最常用的技術是將不可

路由CIDRs 新增至 VPC，並將 VPC CNI 設定為在將 IP 地址配置到 Pod 時使用此額外的 IP 空間。這通常稱為自訂聯網。

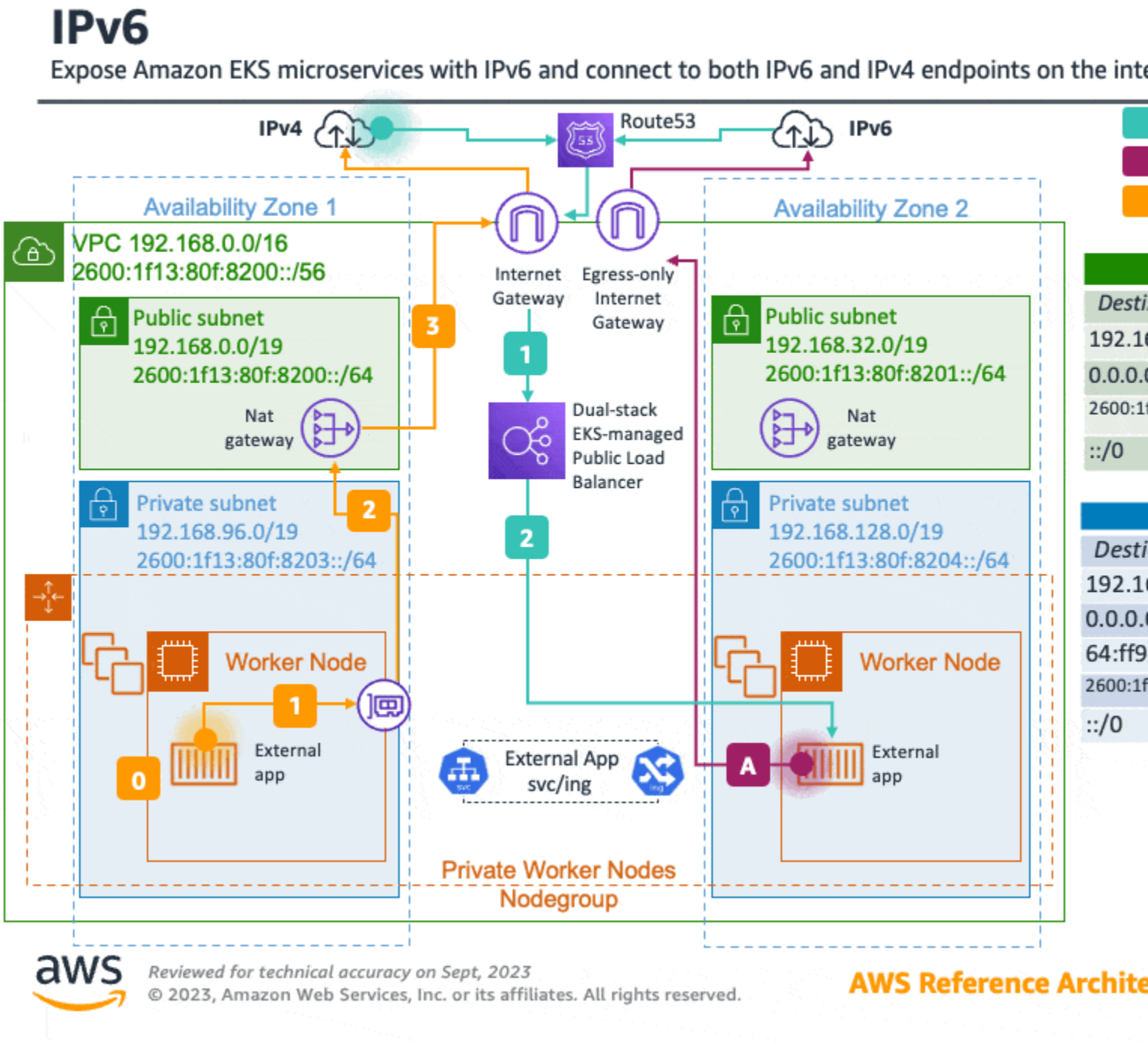
我們將介紹您可以使用哪些 Amazon VPC CNI 變數來最佳化指派給節點的 IPs 暖集區。我們將使用一些非 Amazon EKS 內部但有助於緩解 IP 耗盡的其他架構模式來關閉本節。

使用 IPv6 （建議）

採用 IPv6 是解決 RFC1918 限制的最簡單方法；強烈建議您在選擇網路架構時考慮採用 IPv6 作為您的第一個選項。IPv6 提供更大的總 IP 地址空間，而叢集管理員可以專注於遷移和擴展應用程式，而無需努力解決 IPv4 限制。

Amazon EKS 叢集同時支援 IPv4 和 IPv6。根據預設，EKS 叢集會使用 IPv4 地址空間。在叢集建立時指定 IPv6 型地址空間，將可使用 IPv6。在 IPv6 EKS 叢集中，Pod 和服務會接收 IPv6 地址，同時維持舊版 IPv4 端點連線到 IPv6 叢集上執行之服務的能力，反之亦然。叢集內的所有 pod-to-pod 通訊一律會透過 IPv6 進行。在 VPC (/56) 中，IPv6 子網路的 IPv6 CIDR 區塊大小固定為 /64。這提供 2^64（大約 18 百分位數）IPv6 地址，允許 在 EKS 上擴展部署。

如需詳細資訊，請參閱執行 IPv6 EKS 叢集一節，如需實作體驗，請參閱使用 IPv6 實作研討會中了解 Amazon EKS 上的 IPv6 一節。



最佳化 IPv4 叢集中的 IP 消耗

本節專供執行舊版應用程式和/或尚未準備好遷移至 IPv6 的客戶使用。雖然我們鼓勵所有組織盡快遷移到 IPv6，但我們認識到，有些組織可能仍然需要研究替代方法，以使用 IPv4 擴展容器工作負載。因此，我們也將逐步解說架構模式，以最佳化 Amazon EKS 叢集的 IPv4 (RFC1918) 地址空間使用。

規劃成長

作為防禦 IP 耗盡的第一道防線，我們強烈建議將您的 IPv4 VPCs 和子網路大小納入增長考量，以防止叢集使用所有可用的 IP 地址。如果子網路沒有足夠的可用 IP 地址，您將無法建立新的 Pod 或節點。

在建置 VPC 和子網路之前，建議從所需的工作負載規模向後工作。例如，當使用 eksctl （在 EKS 上建立和管理叢集的簡單 CLI 工具）/19 子網路建置叢集時，預設會建立。/19 的網路遮罩適用於大多數工作負載類型，允許配置超過 8000 個地址。

> **Important**
>
> 當您調整 VPCs和子網路的大小時，可能會有數個元素 (Pod 和節點除外） 可以使用 IP 地址，例如負載平衡器、RDS 資料庫和其他 Vpc 內服務。

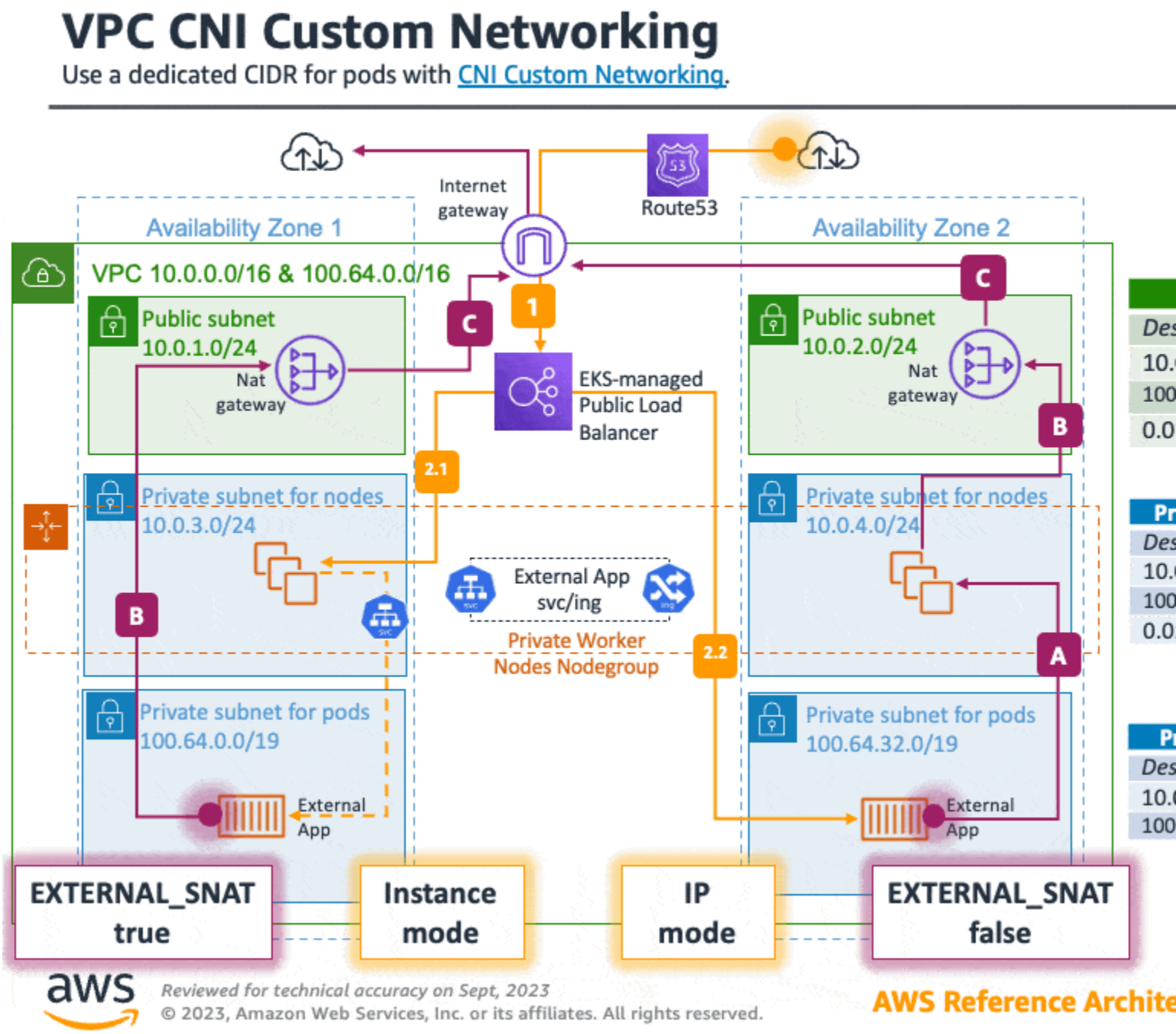
此外，Amazon EKS 可以建立最多 4 個彈性網路介面 (X-ENI)，允許與控制平面通訊 （如需詳細資訊，請參閱此處)。在叢集升級期間，Amazon EKS 會建立新的 X-ENIs並在升級成功時刪除舊的 X-ENI。因此，對於與 EKS 叢集相關聯的子網路，我們建議的網路遮罩至少為 /28 (16 個 IP 地址）。

您可以使用範例 EKS 子網路計算器試算表來規劃您的網路。試算表會根據工作負載和 VPC ENI 組態計算 IP 用量。IP 用量會與 IPv4 子網路進行比較，以判斷組態和子網路大小是否足以滿足您的工作負載。請注意，如果您 VPC 中的子網路用完可用的 IP 地址，建議您使用 VPC 的原始 CIDR 區塊建立新的子網路。請注意，Amazon EKS 現在允許修改叢集子網路和安全群組。

自訂聯網

如果您即將耗盡 RFC1918 IP 空間，您可以使用自訂網路模式， IPs 透過在專用其他子網路中排程 Pod 來節省可路由 IP。雖然自訂聯網會接受次要 CIDR 範圍的有效 VPC 範圍，但我們建議您使用`100.64.0.0/10`共用地址空間 (RFC 6598) 中的 CIDRs，因為相較於 RFC1918 範圍，這些 CIDR 不太可能用於公司設定。例如，您可以使用 `100.64.0.0/16`做為 VPC 的次要 CIDR。如需允許的次要 CIDR 範圍，請參閱 IPv4 CIDR 區塊關聯限制。

如需詳細資訊，請參閱自訂聯網的專用區段。



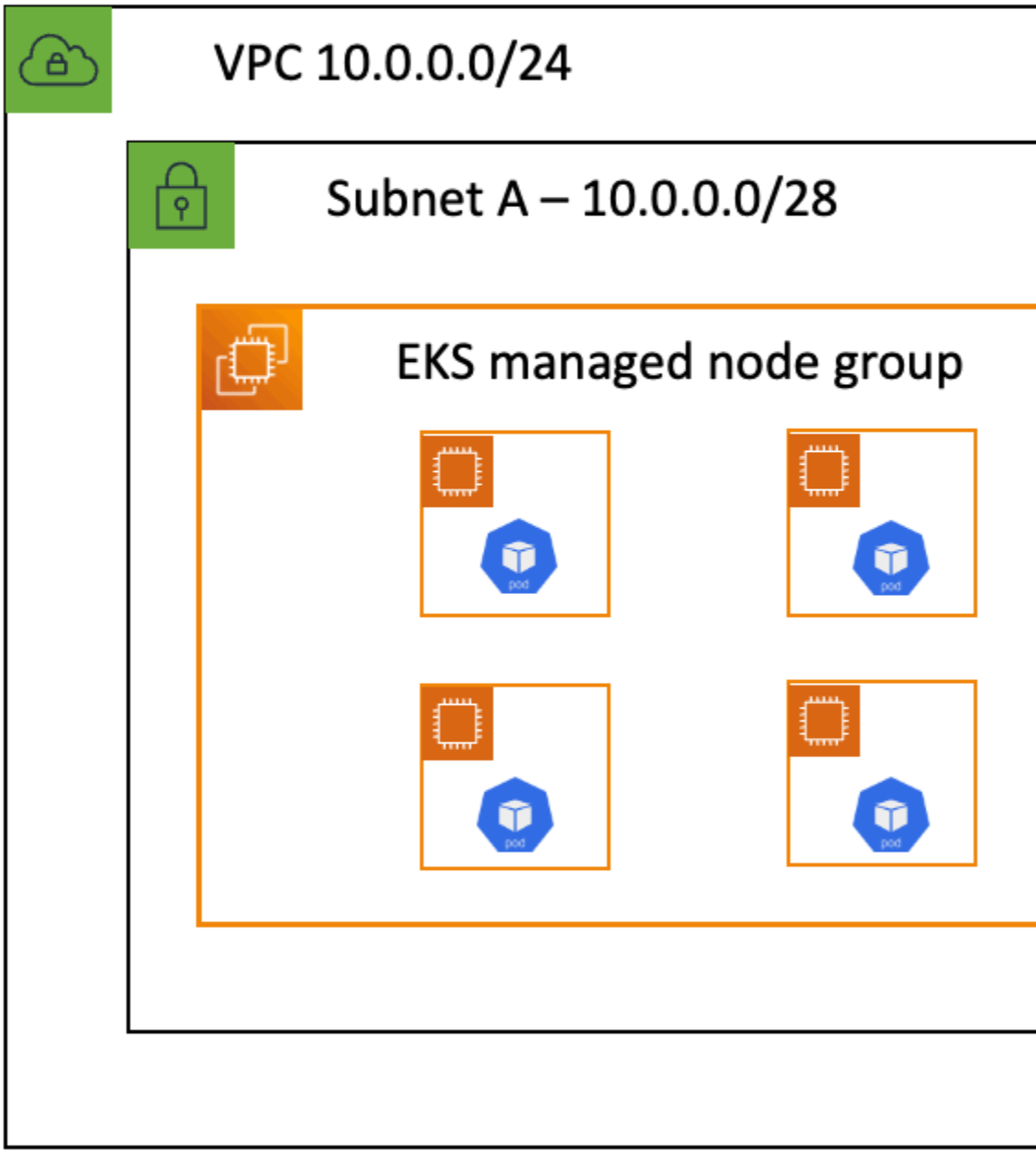
增強型子網路探索

增強型子網路探索透過標記新的子網路，讓 Amazon VPC CNI 可以探索它們，為 IP 耗盡提供簡化的網路組態替代方案。透過增強型子網路探索，目前的工作負載可以繼續在相同的子網路上執行，而 Amazon Elastic Kubernetes Service (Amazon EKS) 現在可以在新的「可用子網路」上排程額外的 Pod。

如果您叢集目前的子網路用完 IP 地址，您可以直接將其他子網路新增至 Amazon EKS 叢集，如下所示：

1. 將新的 CIDR 區塊與您的 VPC 建立關聯。
2. 在新的 CIDR 區塊中建立新的子網路，並使用 "kubernetes.io/role/cni" = "1" 標記它。
3. 啟用 Amazon VPC CNI 附加元件對 "true" 的 ENABLE_SUBNET_DISCOVERY 組態（自 1.18.0 版以來的預設值）。

在 VPC 和 Amazon EKS 叢集上啟用增強型子網路探索後，新的彈性網路界面 (ENIs) 會連接到您的 Amazon EKS 節點，如下圖所述：


VPC 10.0.0.0/24
Subnet A – 10.0.0.0/28
EKS managed node group
pod
pod
pod
pod

如需詳細資訊，請參閱 AWS 容器部落格上的 Amazon VPC CNI 推出增強型子網路探索。

最佳化 IPs暖集區

使用預設組態時，VPC CNI 會將整個 ENI （和相關聯的 IPs) 保留在暖集區中。這可能會耗用大量 IPs，特別是在較大的執行個體類型上。

如果您的叢集子網路可用 IP 地址數量有限，請仔細檢查這些 VPC CNI 組態環境變數：

- `WARM_IP_TARGET`
- `MINIMUM_IP_TARGET`
- `WARM_ENI_TARGET`

您可以設定 的值`MINIMUM_IP_TARGET`，以符合您預期在節點上執行的 Pod 數量。這樣做將確保在建立 Pod 時，CNI 可以從暖集區指派 IP 地址，而無需呼叫 EC2 API。

請注意，將值設定為`WARM_IP_TARGET`太低，將導致對 EC2 API 的額外呼叫，這可能會導致請求限流。對於大型叢集，請使用 搭配 `MINIMUM_IP_TARGET`，以避免調節請求。

若要設定這些選項，您可以下載`aws-k8s-cni.yaml`資訊清單並設定環境變數。撰寫時，最新版本位於此處。檢查組態值的版本是否符合安裝的 VPC CNI 版本。

> **Warning**
>
> 當您更新 CNI 時，這些設定將重設為預設值。請在更新 CNI 之前進行備份。檢閱組態設定，判斷是否需要在更新成功後重新套用。

您可以即時調整 CNI 參數，而不會讓現有應用程式停機，但您應該選擇可支援可擴展性需求的值。例如，如果您使用的是批次工作負載，建議您更新預設值`WARM_ENI_TARGET`以符合 Pod 擴展需求。`WARM_ENI_TARGET` 設定為高值一律會維持執行大型批次工作負載所需的暖 IP 集區，因此可避免資料處理延遲。

> **Warning**
>
> 改善您的 VPC 設計是對 IP 地址耗盡的建議回應。考慮 IPv6 和次要 CIDRs等解決方案。在排除其他選項之後，調整這些值以將暖 IPs 數量降至最低，應該是暫時的解決方案。錯誤設定這些值可能會干擾叢集操作。在對生產系統進行任何變更之前，請務必檢閱此頁面上的考量事項。

監控 IP 地址庫存

除了上述解決方案之外，也請務必掌握 IP 使用率。您可以使用 CNI 指標協助程式監控子網路的 IP 地址庫存。一些可用的指標包括：

- 叢集可支援的 ENIs 數量上限
- 已配置ENIs 數量
- 目前指派給 Pod 的 IP 地址數量
- 可用的 IP 地址總數和上限

您也可以設定 CloudWatch 警示，以便在子網路用完 IP 地址時收到通知。

> **Warning**
>
> 確定 VPC CNI 的`DISABLE_METRICS`變數設定為 false。

進一步考量

還有其他非 Amazon EKS 內部的架構模式，可協助 IP 耗盡。例如，您可以最佳化跨 VPCs通訊，或跨多個帳戶共用 VPC，以限制 IPv4 地址配置。

在此處進一步了解這些模式：

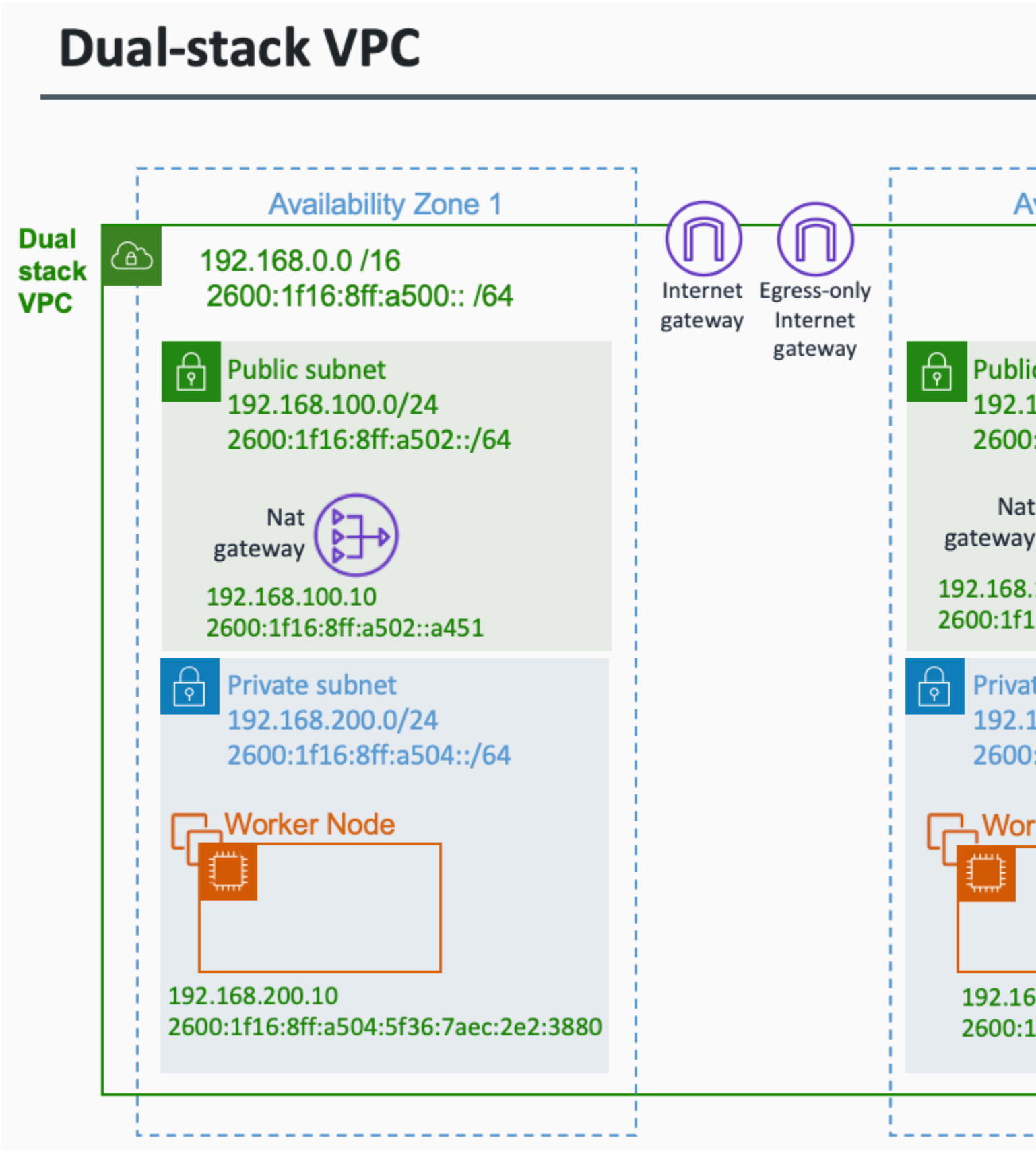
- 設計超大規模 Amazon VPC 網路、
- 使用 Amazon VPC Lattice 建置安全的多帳戶多 VPC 連線。

執行 IPv6 EKS 叢集

IPv6 模式中的 EKS 解決了在大規模 EKS 叢集中常見的 IPv4 耗盡挑戰。EKS 對 IPv6 的支援專注於解決 IPv4 耗盡問題，這源自於 IPv4 地址空間的大小有限。這是我們許多客戶提出的重大考量，與 Kubernetes IPv4/IPv6 雙堆疊功能不同。EKS/IPv6 也將提供使用 IPv6 CIDRs 互連網路邊界的彈性，因此可最大限度地減少遭受 CIDR 重疊的機會，進而解決 2-Fold 問題（叢集內、跨叢集）。在 IPv6 模式 (--ip-family ipv6) 中部署 EKS 叢集時，動作不是可逆的。簡單來說，EKS IPv6 支援會在叢集的整個生命週期內啟用。

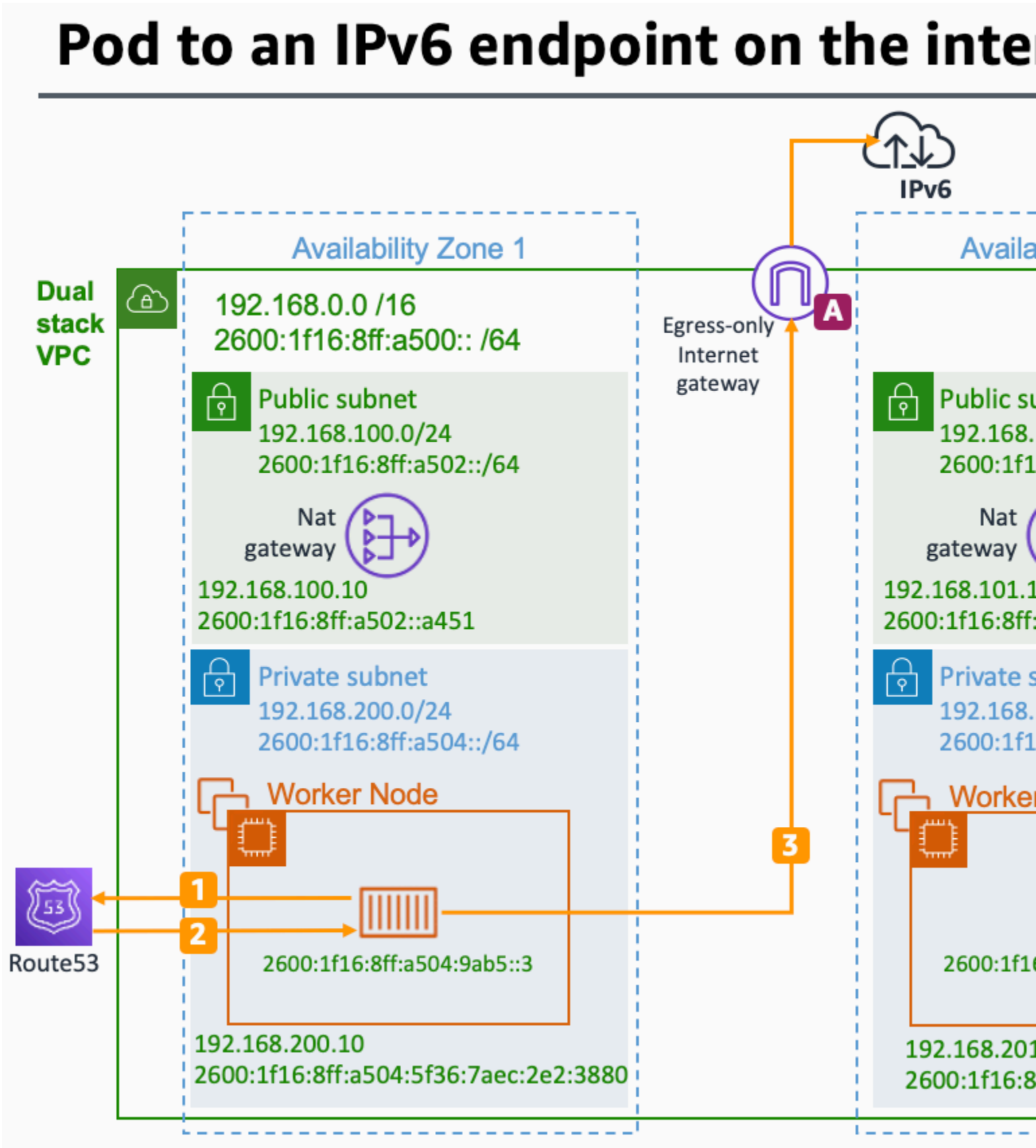
在 IPv6 EKS 叢集中，Pod 和服務將接收 IPv6 地址，同時保持與舊版 IPv4 端點的相容性。這包括外部 IPv4 端點存取叢集內服務的能力，以及 Pod 存取外部 IPv4 端點的能力。

Amazon EKS IPv6 支援利用原生 VPC IPv6 功能。每個 VPC 都會配置 IPv4 地址字首 (CIDR 區塊大小可以是 /16 到 /28)，以及來自 Amazon GUA （全球單點傳送地址） 內唯一的 /56 IPv6 地址字首 （固定）；您可以將 /64 地址字首指派給 VPC 中的每個子網路。IPv4 功能，例如路由表、網路存取控制清單、對等和 DNS 解析，在啟用 IPv6 的 VPC 中以相同的方式運作。然後，VPC 稱為雙堆疊 VPC，遵循雙堆疊子網路，下圖說明支援 EKS/IPV4IPv6IPv6 VPC 基礎模式：


Dual-stack VPC
Availability Zone 1
Dual stack VPC
192.168.0.0 /16
2600:1f16:8ff:a500:: /64
Internet gateway
Egress-only Internet gateway
Public subnet
192.168.100.0/24
2600:1f16:8ff:a502::/64
Nat gateway
192.168.100.10
2600:1f16:8ff:a502::a451
Private subnet
192.168.200.0/24
2600:1f16:8ff:a504::/64
Worker Node
192.168.200.10
2600:1f16:8ff:a504:5f36:7aec:2e2:3880

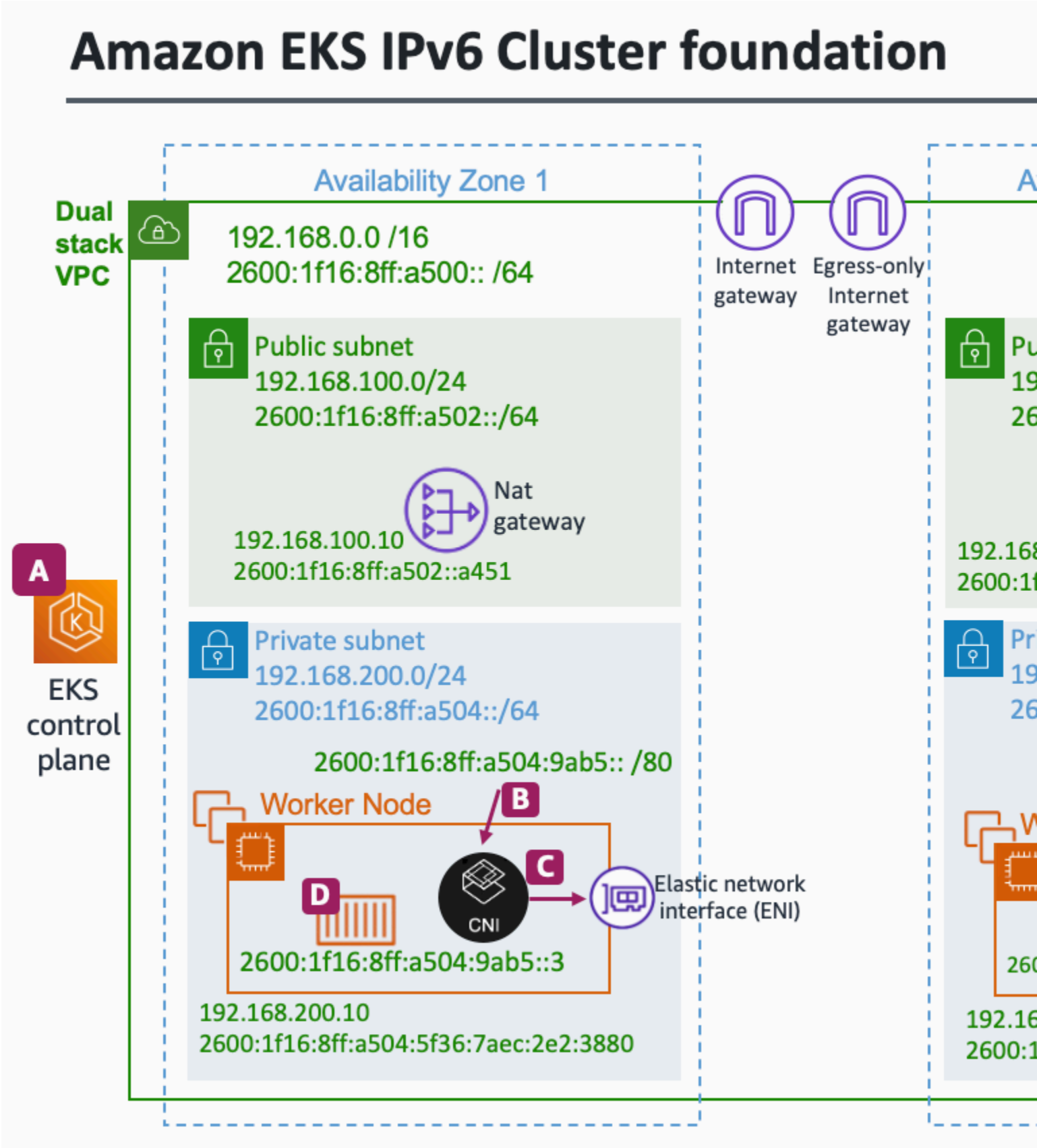
在 IPv6 世界中，每個地址都是網際網路路由。根據預設，VPC 會從公有 GUA 範圍配置 IPv6 CIDR。不過，自 2024 年 8 月起，您也可以將私有 IPv6 定址用於具有 Amazon VPCs IP Address Manager (IPAM) 的 VPC 和子網路。如需詳細資訊，請參閱此 AWS 網路部落格文章和 VPC 文件。

下圖說明 EKS/IPv6 叢集內的 Pod IPv6 網際網路輸出流程：IPv6


Pod to an IPv6 endpoint on the inte
IPv6
Availability Zone 1
Dual stack VPC
192.168.0.0 /16
2600:1f16:8ff:a500:: /64
Egress-only Internet gateway
A
Public subnet
192.168.100.0/24
2600:1f16:8ff:a502::/64
Nat gateway
192.168.100.10
2600:1f16:8ff:a502::a451
Private subnet
192.168.200.0/24
2600:1f16:8ff:a504::/64
Worker Node
1
2
3
Route53
2600:1f16:8ff:a504:9ab5::3
192.168.200.10
2600:1f16:8ff:a504:5f36:7aec:2e2:3880

您可以在 VPC 使用者指南中找到實作 IPv6 子網路的最佳實務。

在 IPv6 EKS 叢集中，節點和 Pod 會收到公有 IPv6 地址。EKS 會根據唯一本機 IPv6 單點傳送地址 (ULA) 將 IPv6 地址指派給服務。IPv6 叢集的 ULA 服務 CIDR 會在叢集建立階段自動指派，而且無法指定，與 IPv4 不同。下圖說明以 EKS/IPv6 為基礎的叢集控制平面資料計劃基礎模式：


Amazon EKS IPv6 Cluster foundation
Availability Zone 1
Dual stack VPC
192.168.0.0 /16
2600:1f16:8ff:a500:: /64
Internet gateway
Egress-only Internet gateway
Public subnet
192.168.100.0/24
2600:1f16:8ff:a502::/64
Nat gateway
192.168.100.10
2600:1f16:8ff:a502::a451
A
EKS control plane
Private subnet
192.168.200.0/24
2600:1f16:8ff:a504::/64
2600:1f16:8ff:a504:9ab5:: /80
Worker Node
B
C
D
CNI
Elastic network interface (ENI)
2600:1f16:8ff:a504:9ab5::3
192.168.200.10
2600:1f16:8ff:a504:5f36:7aec:2e2:3880

概觀

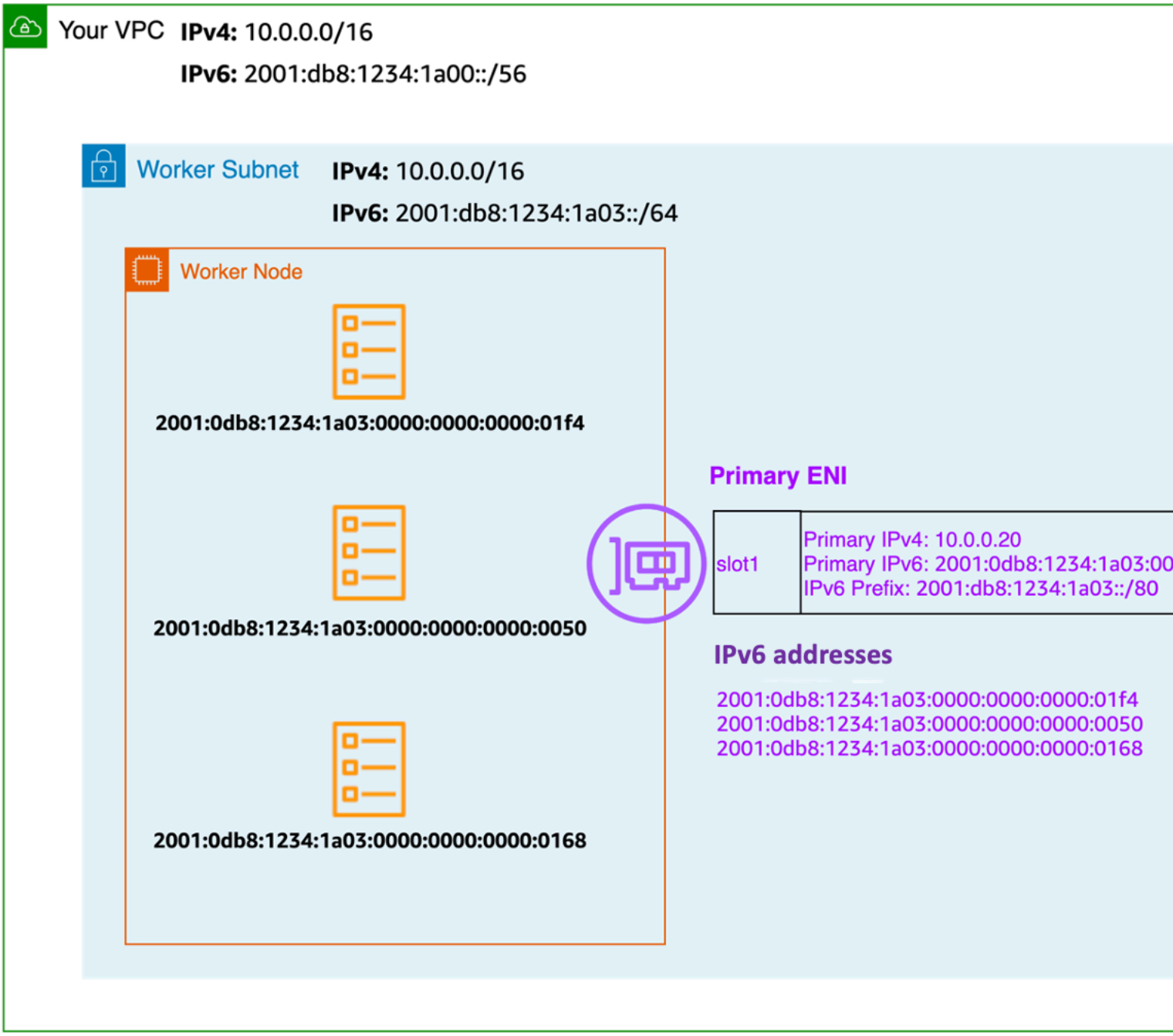
EKS/IPv6 僅支援字首模式 (VPC-CNI 外掛程式 ENI IP 指派模式）。進一步了解字首模式。

字首指派僅適用於 Nitro 型 EC2 執行個體，因此只有在叢集資料平面使用 EC2 Nitro 型執行個體時，才支援 EKS/IPv6。

簡單來說，IPv6 字首 /80 （每個工作者節點） 會產生 ~10^14 個 IPv6 地址，限制因素將不再是 IPs而是 Pod 密度 （依資源）。

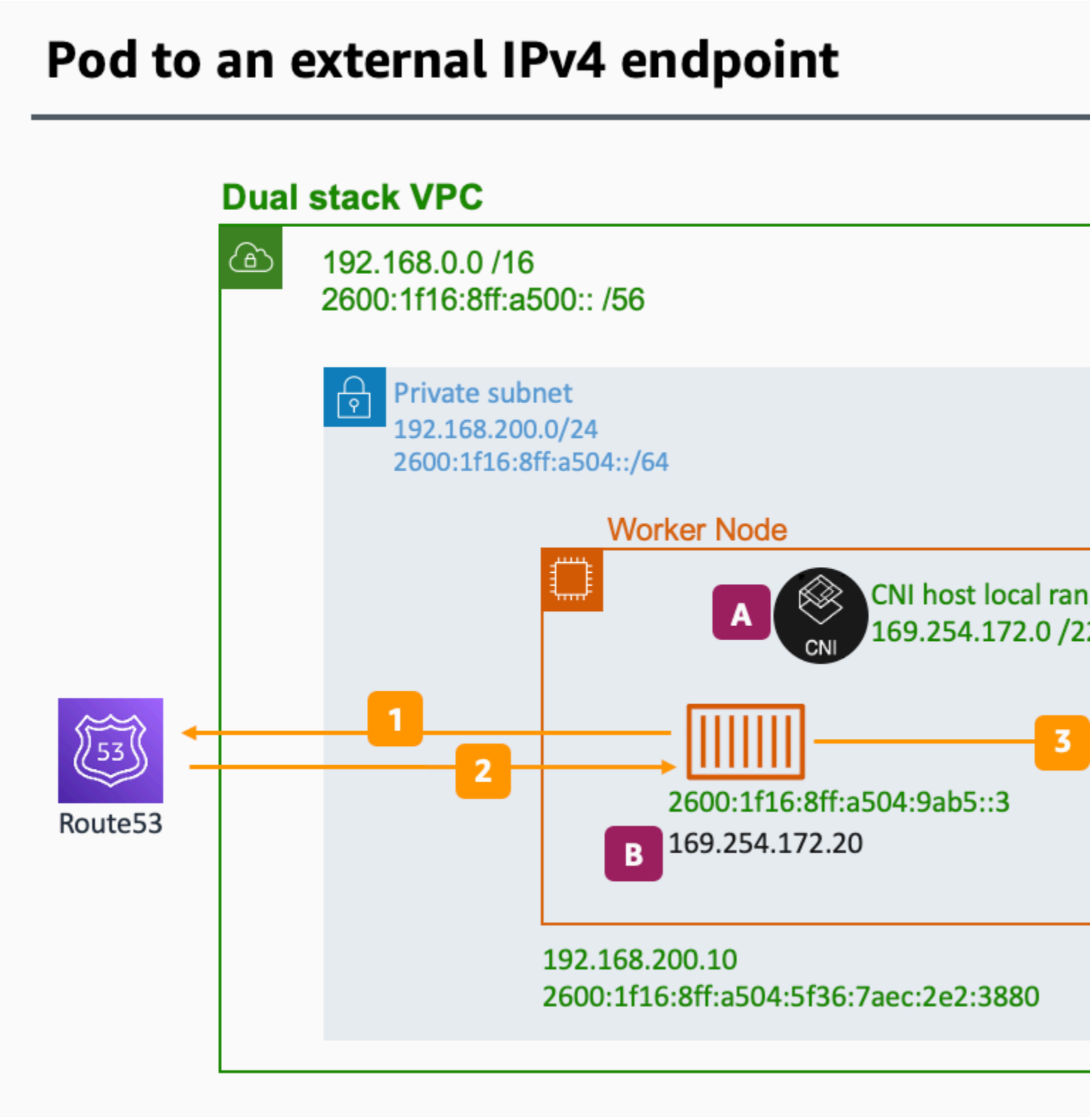
IPv6 字首指派只會在 EKS 工作者節點引導時間進行。此行為已知可以緩解高 Pod 流失 EKS/IPv4 叢集經常因 VPC CNI 外掛程式 (ipamd) 產生的限流 API 呼叫而延遲 Pod 排程的情況，旨在及時配置私有 IPv4 地址。也已知會使 VPC-CNI 外掛程式進階旋鈕不必要地調校 WARM_IP/ENI、MINIMUM_IP。

下圖會縮放至 IPv6 工作者節點彈性網路界面 (ENI)：



每個 EKS 工作者節點都會指派 IPv4 和 IPv6 地址，以及對應的 DNS 項目。對於指定的工作者節點，只會使用來自雙堆疊子網路的單一 IPv4 地址。EKS 對 IPv6 的支援可讓您透過高度評價的輸出限定 IPv4 模型，與 IPv4 端點 (AWS、內部部署、網際網路) 通訊。 IPv4 EKS 實作主機本機 CNI 外掛程式，次要於 VPC CNI 外掛程式，它會配置和設定 Pod 的 IPv4 地址。CNI 外掛程式為 169.254.172.0/22 範圍的 Pod 設定主機特定的不可路由 IPv4 地址。指派給 Pod 的 IPv4 地址對 worker-node 是唯一的，不會在 worker-node 之外公告。 169.254.172.0/22 提供最多 1024 個唯一 IPv4 地址，可支援大型執行個體類型。

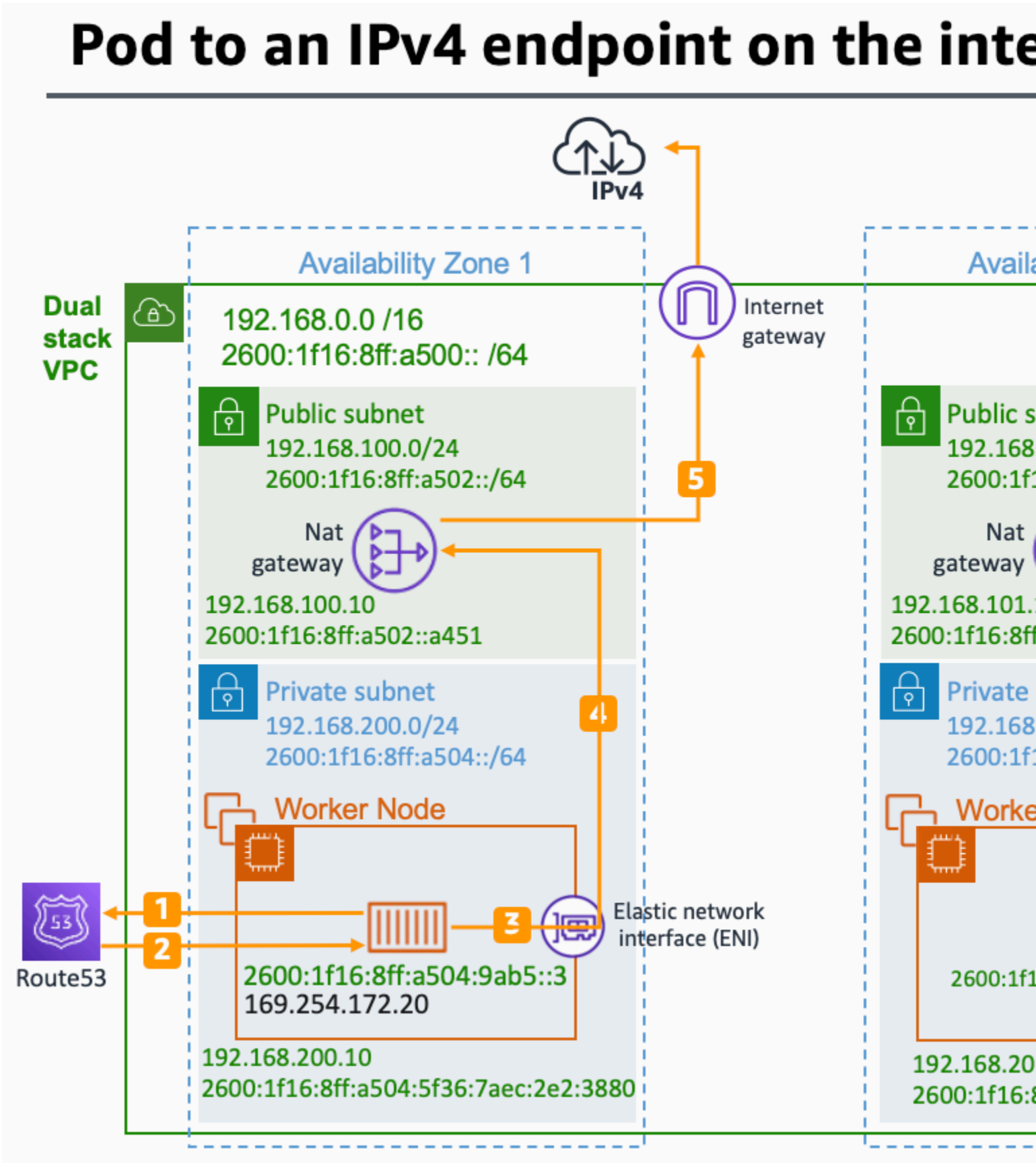
下圖說明連接至叢集邊界 （非網際網路） 外 IPv4 端點的 IPvIPv6 Pod 流程：



在上圖中，Pod 將對端點執行 DNS 查詢，並在收到 IPv4 "A" 回應時，Pod 的節點限定唯一 IPv4 地址會透過來源網路地址轉譯 (SNAT) 轉譯為連接到 EC2 工作者節點之主要網路界面的私有 IPv4 (VPC) 地址。

> **Note**
>
> 上述模式要求在執行 EKS/IPv6 Pod 的子網路上停用 DNS64。IPv6 啟用 DNS64 時，DNS 解析程式會傳回IPv4-only端點的合成 IPv6 地址，以及 IPv4 地址。因此，流量會透過 NAT Gateway 的 NAT64 功能（如果包含在架構中）路由，而不是保留在 VPC 內，如上圖所示。這可能會導致非預期的 NAT Gateway 用量和相關聯的成本。

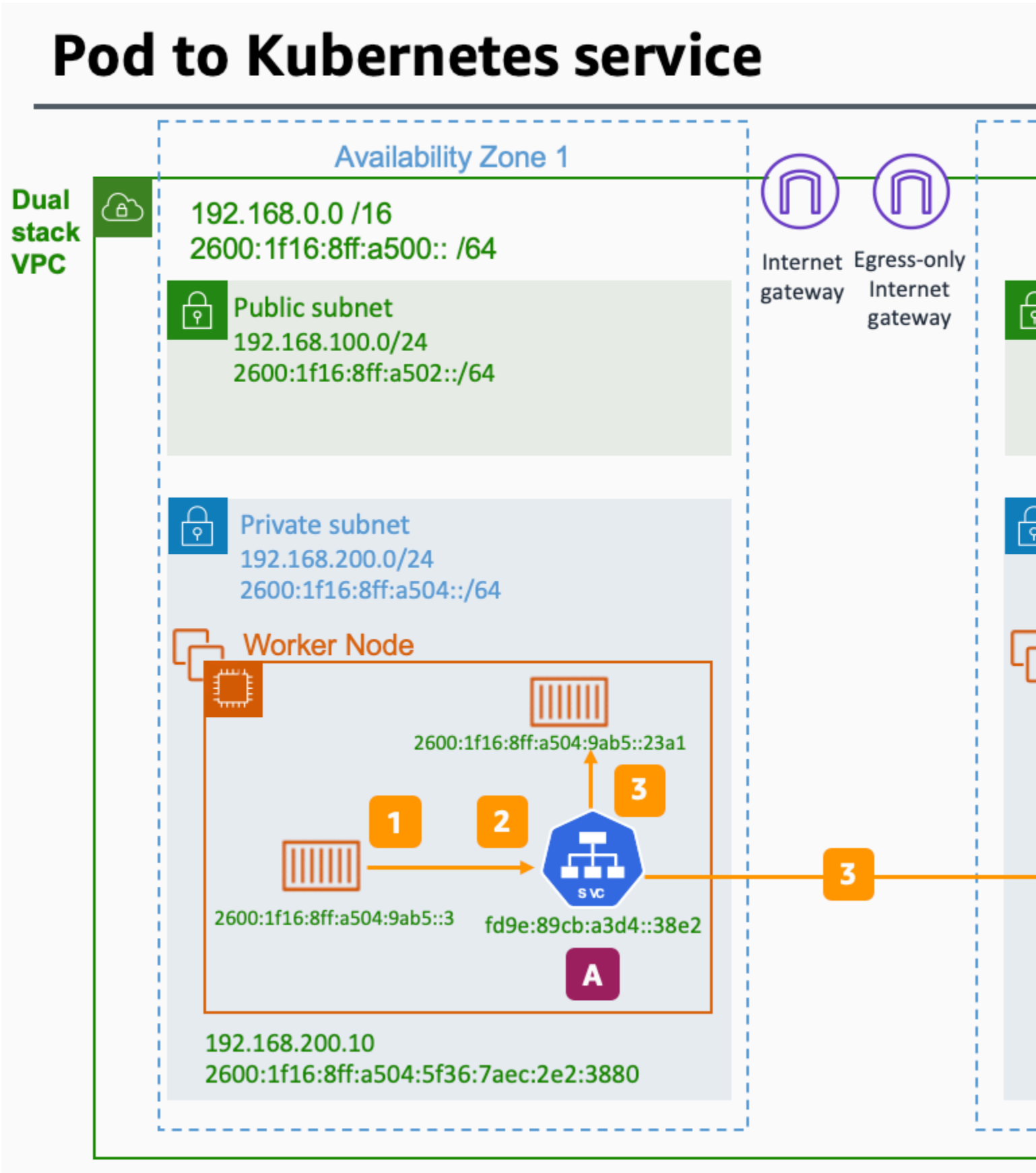
EKS/IPv6 Pod 還需要使用公有 IPv4 地址透過網際網路連線至 IPv4 端點，以實現存在類似的流程。下圖說明連接至叢集邊界外 IPv4 端點的 IPv6 Pod 流程（網際網路可路由）：


Pod to an IPv4 endpoint on the inte
IPv4
Availability Zone 1
Dual stack VPC
192.168.0.0 /16
2600:1f16:8ff:a500:: /64
Internet gateway
Public subnet
192.168.100.0/24
2600:1f16:8ff:a502::/64
Nat gateway
192.168.100.10
2600:1f16:8ff:a502::a451
Private subnet
192.168.200.0/24
2600:1f16:8ff:a504::/64
Worker Node
Route53
Elastic network interface (ENI)
2600:1f16:8ff:a504:9ab5::3
169.254.172.20
192.168.200.10
2600:1f16:8ff:a504:5f36:7aec:2e2:3880
1
2
3
4
5

在上圖中，Pod 將對端點執行 DNS 查詢，並在收到 IPv4 "A" 回應時，Pod 的節點限定唯一 IPv4 地址會透過來源網路地址轉譯 (SNAT) 轉譯為連接到 EC2 工作者節點之主要網路界面的私有 IPv4 (VPC) 地址。然後，Pod IPv4 地址 （ 來源 IPv4：EC2 主要 IP) 會路由至IPv4 EC2 主要 IP (SNAT) 轉譯為有效網際網路可路由 IPv4 公有 IP 地址 (NAT 閘道指派的公有 IP) 的 IPv4 NAT 閘道。

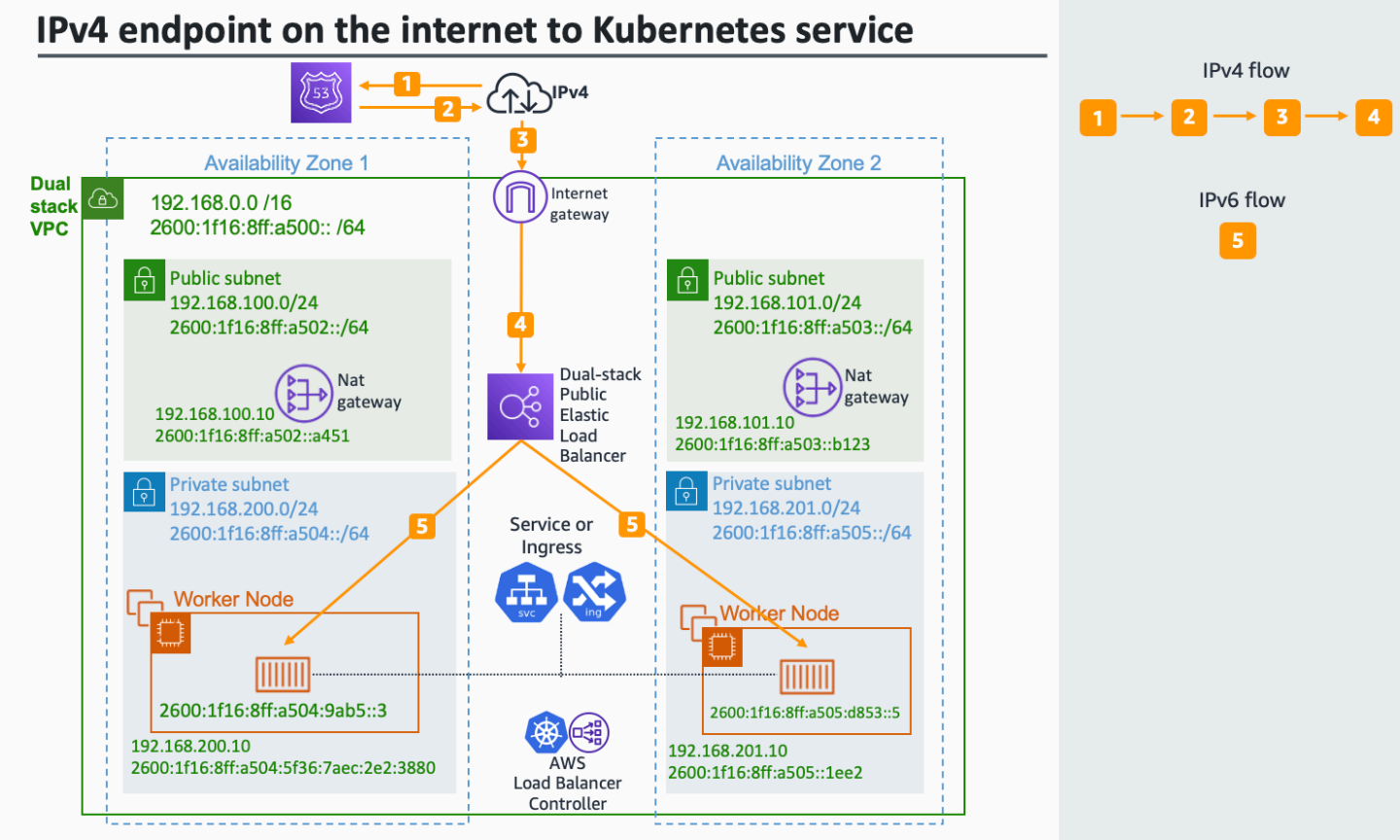
節點之間的任何 Pod-to-Pod 通訊一律使用 IPv6 地址。VPC CNI 會設定 iptable 來處理 IPv6，同時封鎖任何 IPv4 連線。

Kubernetes 服務只會從唯一本機 IPv6 單點傳送地址 (ULA) 接收 IPv6 地址 (ClusterIP)。 IPv6 IPv6 叢集的 ULA 服務 CIDR 會在 EKS 叢集建立階段自動指派，且無法修改。下圖說明 Pod 到 Kubernetes Service 的流程：


Pod to Kubernetes service
Availability Zone 1
Dual stack VPC
192.168.0.0 /16
2600:1f16:8ff:a500:: /64
Internet gateway
Egress-only Internet gateway
Public subnet
192.168.100.0/24
2600:1f16:8ff:a502::/64
Private subnet
192.168.200.0/24
2600:1f16:8ff:a504::/64
Worker Node
2600:1f16:8ff:a504:9ab5::23a1
1
2
3
3
SVC
2600:1f16:8ff:a504:9ab5::3
fd9e:89cb:a3d4::38e2
A
192.168.200.10
2600:1f16:8ff:a504:5f36:7aec:2e2:3880

服務會使用 AWS 負載平衡器公開至網際網路。負載平衡器會接收公有 IPv4 和 IPv6 地址，即 a.k.a 雙堆疊負載平衡器。對於存取 IPv6 叢集 kubernetes 服務的 IPv4 用戶端，負載平衡器會執行 IPv4 到 IPv6 轉譯。

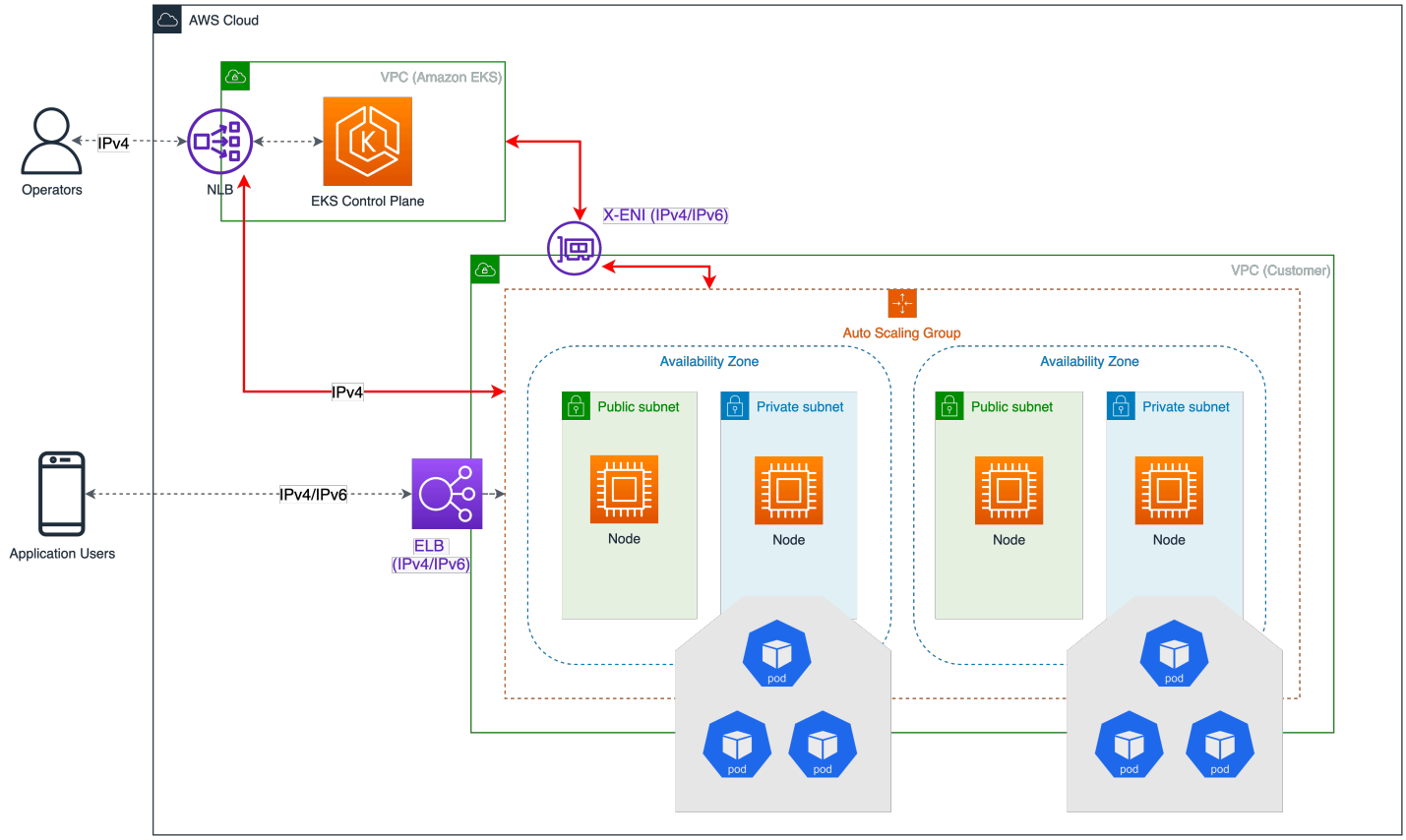
Amazon EKS 建議在私有子網路中執行工作者節點和 Pod。您可以在公有子網路中建立公有負載平衡器，以平衡私有子網路中節點上執行之 Pod 的流量。下圖說明存取 EKS/IPv6 輸入型服務的網際網路 IPv4 使用者：



> **Note**
>
> 上述模式需要部署最新版本的 AWS 負載平衡器控制器

EKS 控制平面資料平面通訊

EKS 將以雙堆疊模式 (IPv4/IPv6) 佈建跨帳戶 ENIs (X-ENIs)。kubelet 和 kube-proxy 等 Kubernetes 節點元件已設定為支援雙堆疊。Kubelet 和 kube-proxy 在 hostNetwork 模式下執行，並繫結至連接到節點主要網路界面的 IPv4 和 IPv6 地址。Kubernetes api-server 透過以 IPv6 為基礎的 X-ENIs 與 Pod 和節點元件通訊。Pod 透過 X-ENIs 與 api-server 通訊，Pod 與 api-server 通訊一律使用 IPv6 模式。



建議

根據運算資源排程

單一 IPv6 字首足以在單一節點上執行許多 Pod。這也會有效地移除節點上 Pod 數目上限的 ENI 和 IP 限制。雖然 IPv6 會移除對 max-Pod 的直接相依性，但使用具有 m5.large 等較小執行個體類型的字首附件時，您可能會在用盡執行個體的 CPU 和記憶體資源之前，長時間用盡執行個體的 CPU 和記憶體資源。如果您使用自我管理節點群組或具有自訂 AMI ID 的受管節點群組，則必須手動設定 EKS 建議的最大 Pod 值。

您可以使用下列公式來判斷您可以在 IPv6 EKS 叢集的節點上部署的 Pod 數量上限。

```
((Number of network interfaces for instance type (number of prefixes per network
 interface-1)* 16) + 2
```

```
((3 ENIs)_((10 secondary IPs per ENI-1)_ 16)) + 2 = 460 (real)
```

受管節點群組會自動為您計算 Pod 的數量上限。避免變更最大 Pod 數量的 EKS 建議值，以避免由於資源限制而導致 Pod 排程失敗。

評估現有自訂聯網的目的

如果目前已啟用自訂聯網，Amazon EKS 建議使用 IPv6 重新評估您對其的需求。如果您選擇使用自訂聯網來解決 IPv4 耗盡問題，就不再需要使用 IPv6。如果您使用自訂聯網來滿足安全需求，例如節點和 Pod 的個別網路，建議您提交 EKS 藍圖請求。

EKS/IPv6 叢集中的 Fargate Pod

EKS 支援在 Fargate 上執行之 Pod 的 IPv6。在 Fargate 上執行的 Pod 將使用從 VPC CIDR 範圍 (IPv4IPv6) 雕刻的 IPv6 和 VPC 可路由私有 IPv4 地址。簡單來說，您的 EKS/Fargate Pods 叢集寬密度將限制為可用的 IPv4 和 IPv6 地址。建議調整雙堆疊子網路/VPC CIDRs的大小，以因應未來成長。如果基礎子網路不包含可用的 IPv4 地址，無論 IPv6 可用的地址為何，您將無法排程新的 Fargate Pod。

部署 AWS Load Balancer 控制器 (LBC)

上游樹狀內 Kubernetes 服務控制器不支援 IPv6。建議使用最新版本的 AWS Load Balancer 控制器附加元件。LBC 只會在使用對應的 kubernetes 服務/輸入定義時部署雙堆疊 NLB 或雙堆疊 ALB，並標註：`"alb.ingress.kubernetes.io/ip-address-type: dualstack"` 和 `"alb.ingress.kubernetes.io/target-type: ip"`

自訂聯網

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

根據預設，Amazon VPC CNI 會將從主要子網路選取的 IP 地址指派給 Pod。主要子網路是主要 ENI 連接的子網路 CIDR，通常是節點/主機的子網路。

如果子網路 CIDR 太小，CNI 可能無法取得足夠的次要 IP 地址來指派給您的 Pod。這是 EKS IPv4 叢集的常見挑戰。

自訂聯網是此問題的一個解決方案。

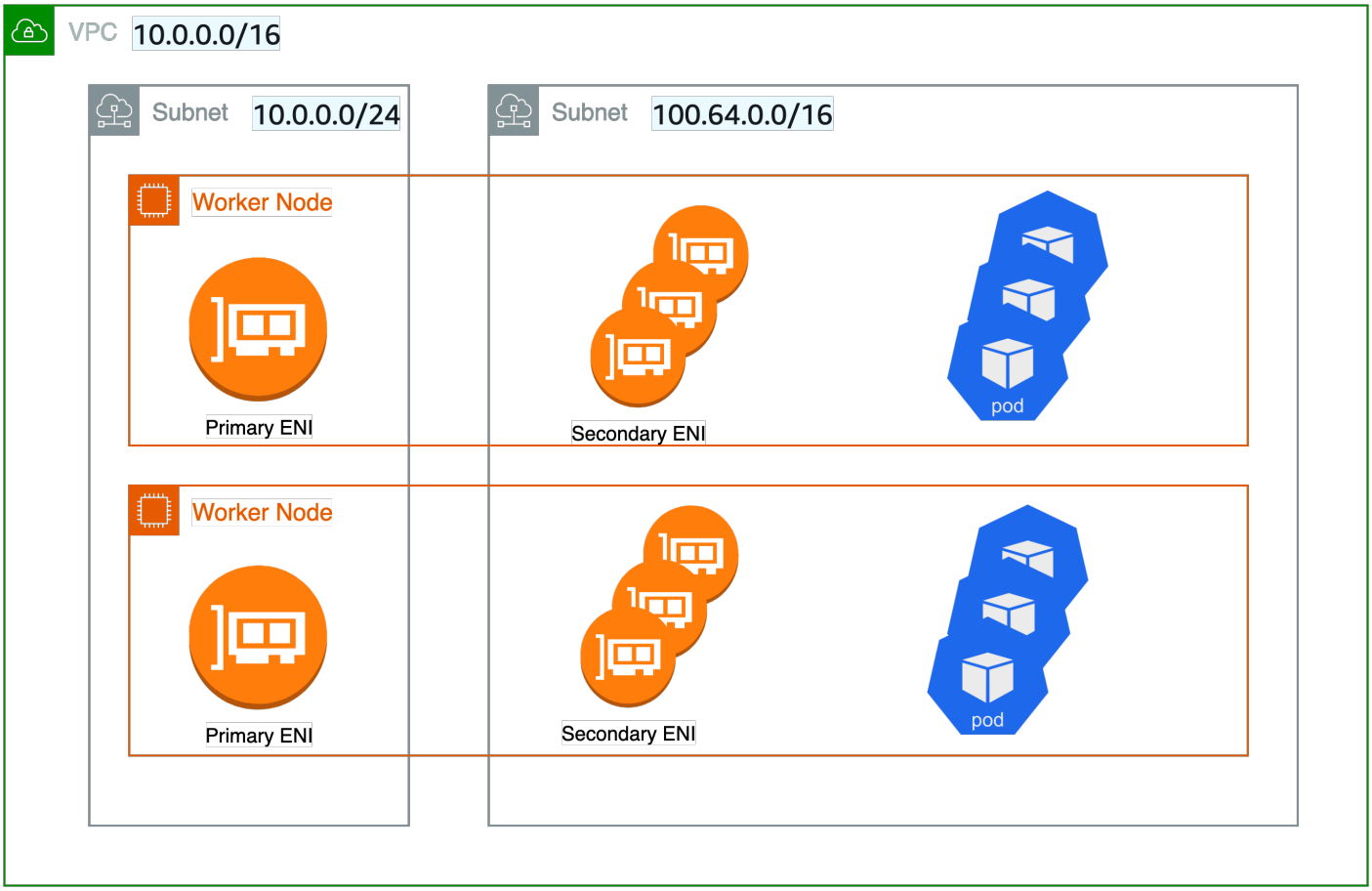
自訂聯網透過從次要 VPC 地址空間 (CIDR) 指派節點和 Pod IPs來解決 IP 耗盡問題。自訂聯網支援 ENIConfig 自訂資源。ENIConfig 包含備用子網路 CIDR 範圍 （從次要 VPC CIDR 取得），以及 Pod 所屬的安全群組。啟用自訂聯網時，VPC CNI 會在 ENIs ENIConfig。CNI 會從 ENIConfig CRD 中定義的 CIDR 範圍指派 IP 地址給 Pod。

由於自訂聯網不使用主要 ENI，因此您可以在節點上執行的 Pod 數量上限較低。主機網路 Pod 會繼續使用指派給主要 ENI 的 IP 地址。此外，主要 ENI 用於處理來源網路轉譯，並將 Pod 流量路由到節點外部。

範例組態

雖然自訂聯網會接受次要 CIDR 範圍的有效 VPC 範圍，但我們建議您使用`100.64.0.0/10`共用地址空間 (RFC 6598) 中的 CIDRs，因為相較於其他 RFC1918 範圍，這些 CIDR 不太可能用於公司設定。例如，您可以使用 `100.64.0.0/16`做為 VPC 的次要 CIDR。如需有關可搭配 VPC 使用的允許和限制 CIDR 區塊關聯的其他資訊，請參閱 VPC 文件的 VPC 和子網路大小一節中的 IPv4 CIDR 區塊關聯限制。

如下圖所示，工作者節點的主要彈性網路界面 (ENI) 仍然使用主要 VPC CIDR 範圍（在此案例中為 10.0.0.0/16)，但次要 ENIs 使用次要 VPC CIDR 範圍（在此案例中為 100.64.0.0/16)。現在，若要讓 Pod 使用 100.64.0.0/16 CIDR 範圍，您必須將 CNI 外掛程式設定為使用自訂聯網。您可以遵循此處所述的步驟。



如果您希望 CNI 使用自訂聯網，請將AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG環境變數設定為true。

```
kubectl set env daemonset aws-node -n kube-system
 AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true
```

當 時AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true，CNI 會從 中定義的子網路指派 Pod IP 地址ENIConfig。ENIConfig 自訂資源用於定義將排定 Pod 的子網路。

```
apiVersion : crd.k8s.amazonaws.com/v1alpha1
kind : ENIConfig
metadata:
  name: us-west-2a
spec:
  securityGroups:
    - sg-0dff111a1d11c1c11
  subnet: subnet-011b111c1f11fdf11
```

建立ENIconfig自訂資源時，您需要建立新的工作者節點並耗盡現有的節點。現有的工作者節點和 Pod 將保持不變。

建議

使用自訂聯網時

如果您正在處理 IPv4 耗盡，而且還無法使用 IPv6，建議您考慮自訂聯網。Amazon EKS 對 RFC6598 空間的支援可讓您擴展 Pod，超越 RFC1918 來解決耗盡挑戰。請考慮搭配自訂聯網使用字首委派，以增加節點上的 Pods 密度。

如果您有在具有不同安全群組需求的不同網路上執行 Pod 的安全需求，您可以考慮自訂聯網。啟用自訂聯網時，Pod 會使用與節點的主要網路界面不同的子網路或安全群組，如 ENIConfig 中所定義。

自訂聯網確實是部署多個 EKS 叢集和應用程式以連接內部部署資料中心服務的理想選項。您可以為 Amazon Elastic Load Balancing 和 NAT-GW 等服務增加 VPC 中的 EKS 可存取的私有地址 (RFC1918) 數量，同時為多個叢集的 Pod 使用不可路由的 CG-NAT 空間。使用傳輸閘道和共用服務 VPC 的自訂聯網（包括跨多個可用區域的 NAT 閘道以實現高可用性）可讓您提供可擴展且可預測的流量流程。此部落格文章描述一種架構模式，這是使用自訂聯網將 EKS Pod 連接到資料中心網路的最推薦方法之一。

避免自訂聯網

準備好實作 IPv6

自訂聯網可以緩解 IP 耗盡問題，但需要額外的操作開銷。如果您目前正在部署雙堆疊 (IPv4/IPv6) VPC，或您的計劃包含 IPv6 支援，建議您改為實作 IPv6 叢集。您可以設定 IPv6 EKS 叢集並遷移您的應用程式。在 IPv6 EKS 叢集中，Kubernetes 和 Pod 都會取得 IPv6 地址，並且可以與 IPv4 和 IPv6 端點進行傳入和傳出通訊。請檢閱執行 IPv6 EKS 叢集的最佳實務。

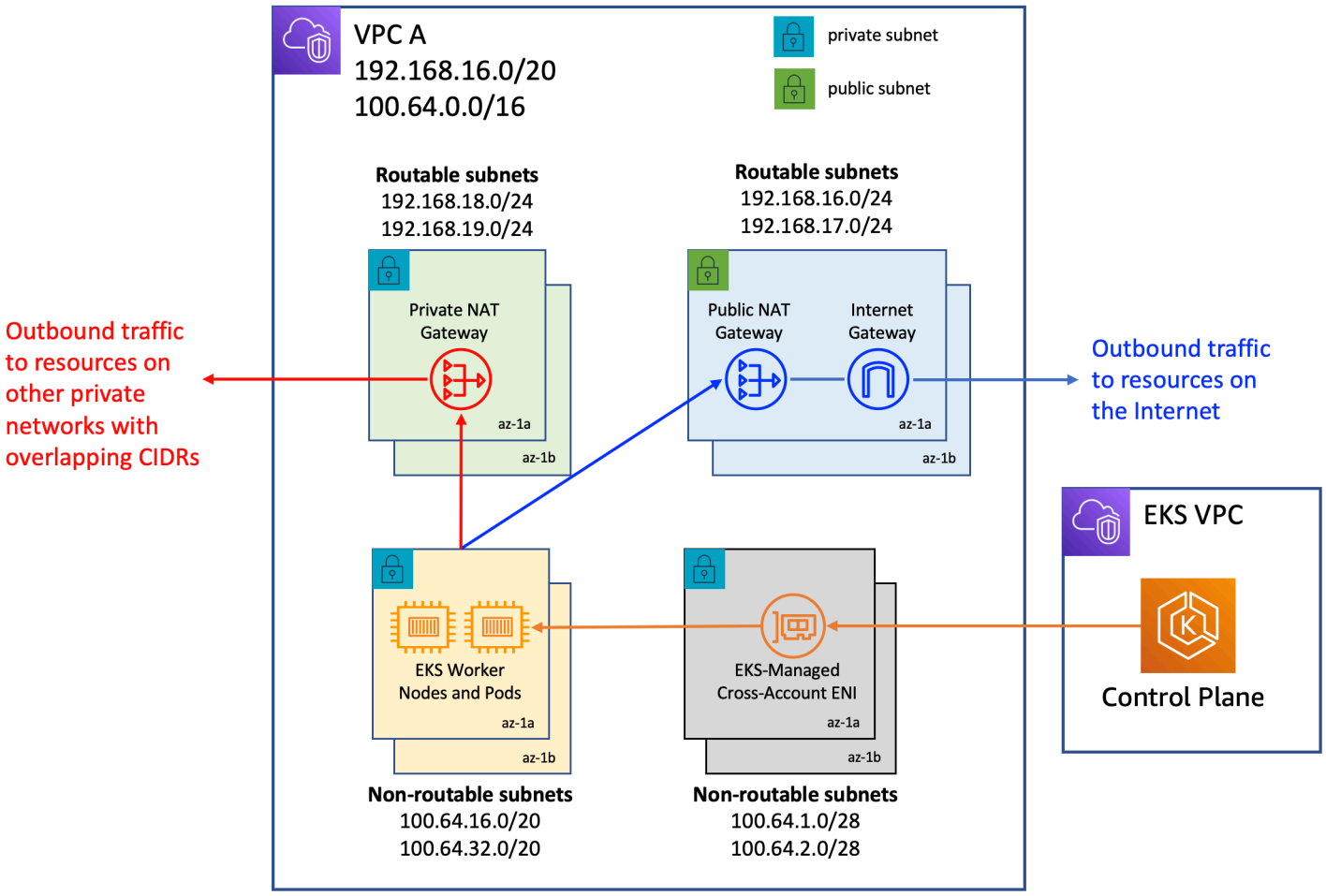
耗盡的 CG-NAT 空間

此外，如果您目前正在使用來自 CG-NAT 空間的 CIDRs，或無法將次要 CIDR 與叢集 VPC 連結，您可能需要探索其他選項，例如使用替代的 CNI。我們強烈建議您取得商業支援或擁有內部知識，以偵錯並將修補程式提交至開放原始碼 CNI 外掛程式專案。如需詳細資訊，請參閱替代 CNI 外掛程式使用者指南。

使用私有 NAT 閘道

Amazon VPC 現在提供私有 NAT 閘道功能。Amazon 的私有 NAT Gateway 可讓私有子網路中的執行個體連線至具有重疊 CIDRs 的其他 VPCs 和內部部署網路。考慮利用本部落格文章中所述的方法，採用私有 NAT 閘道來克服因 CIDRs 重疊而導致的 EKS 工作負載的通訊問題，這是由用戶端表達的重大投訴。自訂聯網無法自行解決重疊的 CIDR 困難，而且會增加組態挑戰。

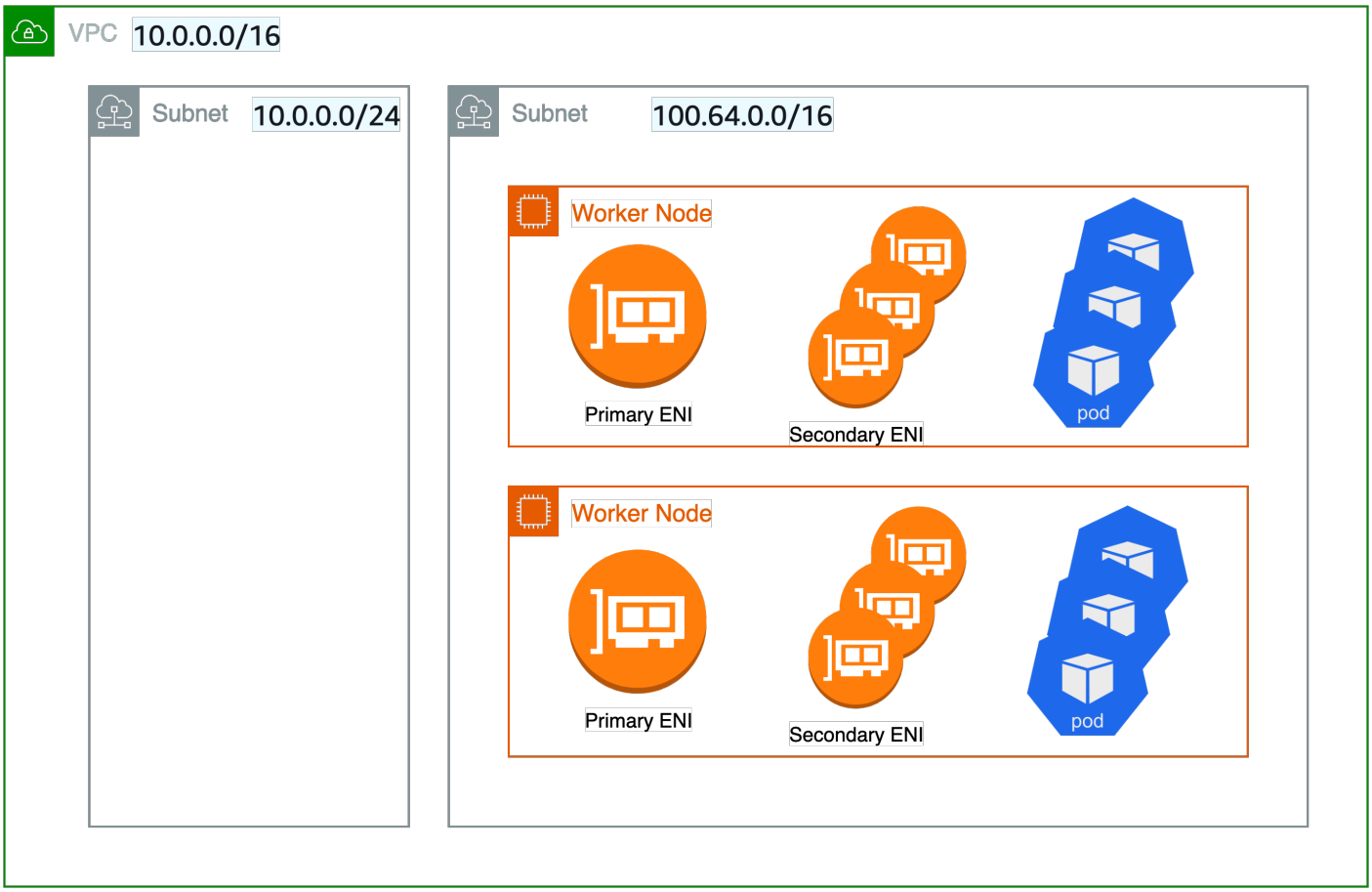
此部落格文章實作中使用的網路架構遵循 Amazon VPC 文件中啟用重疊網路之間的通訊建議。如本部落格文章所示，您可以搭配 RFC6598 地址來擴展私有 NAT Gateway 的使用，以管理客戶的私有 IP 耗盡問題。EKS 叢集、工作者節點部署在不可路由的 100.64.0.0/16 VPC 次要 CIDR 範圍內，而私有 NAT 閘道 NAT 閘道則部署到可路由的 RFC1918 CIDR 範圍。部落格說明如何使用傳輸閘道來連接 VPCs，以促進具有重疊不可路由 CIDR 範圍VPCs 之間的通訊。對於 VPC 不可路由地址範圍內的 EKS 資源需要與其他沒有重疊地址範圍VPCs 通訊的使用案例，客戶可以選擇使用 VPC 對等互連來互連此類 VPCs。此方法可以節省潛在成本，因為現在可透過 VPC 互連連線在可用區域內傳輸所有資料都是免費的。



節點和 Pod 的唯一網路

基於安全考量，如果您需要將節點和 Pod 隔離到特定網路，建議您從較大的次要 CIDR 區塊 （例如 100.64.0.0/8) 將節點和 Pod 部署到子網路。在 VPC 中安裝新的 CIDR 之後，您可以使用次要 CIDR 部署另一個節點群組，並耗盡原始節點，以自動將 Pod 重新部署到新的工作者節點。如需如何實作此項目的詳細資訊，請參閱此部落格文章。

下圖所示的設定不會使用自訂聯網。相反地，Kubernetes 工作者節點部署在 VPC 次要 VPC CIDR 範圍的子網路上，例如 100.64.0.0/10。您可以讓 EKS 叢集持續執行 （控制平面會保留在原始子網路上），但節點和 Pod 會移至次要子網路。這是另一個非傳統技術，可減輕 VPC 中 IP 耗盡的風險。我們建議先耗盡舊節點，再將 Pod 重新部署到新的工作者節點。



使用可用區域標籤自動化組態

您可以讓 Kubernetes 自動為工作者節點可用區域 (AZ) 套用對應的 ENIConfig。

Kubernetes 會自動將標籤新增至topology.kubernetes.io/zone工作者節點。當您每個可用區域只有一個次要子網路 （替代 CIDR) 時，Amazon EKS 建議使用可用區域做為 ENI 組態名稱。然後，您可以將用於探索 ENI 組態名稱的標籤設定為 topology.kubernetes.io/zone。請注意，標籤failure-domain.beta.kubernetes.io/zone已棄用，並以標籤 取代topology.kubernetes.io/zone。

1. 將 name 欄位設定為 VPC 的可用區域。
2. 透過下列命令啟用自動組態
3. 透過下列命令設定組態標籤

```
kubectl set env daemonset aws-node -n kube-system
 "AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG=true"
```

```
kubectl set env daemonset aws-node -n kube-system
 "ENI_CONFIG_LABEL_DEF=topology.kubernetes.io/zone"
```

如果每個可用區域都有多個次要子網路，則需要建立特定的 ENI_CONFIG_LABEL_DEF。您可以考慮使用自訂 eniConfig 名稱ENI_CONFIG_LABEL_DEF將 設定為 k8s.amazonaws.com/eniConfig和 標籤節點，例如 k8s.amazonaws.com/eniConfig=us-west-2a-subnet-1和 k8s.amazonaws.com/eniConfig=us-west-2a-subnet-2。

設定次要聯網時取代 Pod

啟用自訂聯網不會修改現有的節點。自訂聯網是一種破壞性動作。建議您使用呼叫 Lambda 函數的 AWS CloudFormationhttps://docs.aws.amazon.com/eks/latest/userguide/getting-started.html自訂資源來更新 aws-node Daemonset 與環境變數，以便在佈建工作者節點之前啟用自訂聯網，而不是在啟用自訂聯網之後輪流取代叢集中的所有工作者節點。

如果在切換至自訂 CNI 網路功能之前，叢集中有任何執行中 Pod 的節點，您應該封鎖並耗盡節點，以正常關閉 Pod，然後終止節點。只有符合 ENIConfig 標籤或註釋的新節點會使用自訂聯網，因此在這些新節點上排程的 Pod 可以從次要 CIDR 指派 IP。

計算每個節點的最大 Pod

由於節點的主要 ENI 不再用於指派 Pod IP 地址，因此您可以在指定的 EC2 執行個體類型上執行的 Pod 數量會減少。若要解決此限制，您可以搭配自訂聯網使用字首指派。透過字首指派，每個次要 IP 都會取代為次要 ENIs 上的 /28 字首。

考慮具有自訂聯網的 m5.large 執行個體的 Pod 數量上限。

在沒有字首指派的情況下，您可以執行的 Pod 數目上限為 29

- `3 ENIs - 1) * (10 secondary IPs per ENI - 1 + 2 = 20`

啟用字首附件會將 Pod 數量增加到 290。

- `(3 ENIs - 1) * ((10 secondary IPs per ENI - 1) * 16 + 2 = 290`

不過，我們建議將最大 Pod 設定為 110 而不是 290，因為執行個體具有相當少量的虛擬 CPUs。在較大的執行個體上，EKS 建議最大 Pod 值為 250。使用具有較小執行個體類型的字首附件 （例如 m5.large) 時，您可能會在其 IP 地址之前充分耗盡執行個體的 CPU 和記憶體資源。

> **Note**
>
> 當 CNI 字首將 /28 字首配置給 ENI 時，它必須是 IP 地址的連續區塊。如果產生字首的子網路高度分段，字首連接可能會失敗。您可以為叢集建立新的專用 VPC，或為字首附件預留一組 CIDR，以減輕這種情況。如需此主題的詳細資訊，請造訪子網路 CIDR 保留。

識別 CG-NAT 空間的現有用量

自訂聯網可讓您緩解 IP 耗盡問題，但無法解決所有挑戰。如果您已經為叢集使用 CG-NAT 空間，或只是無法將次要 CIDR 與叢集 VPC 建立關聯，我們建議您探索其他選項，例如使用替代 CNI 或移至 IPv6 叢集。

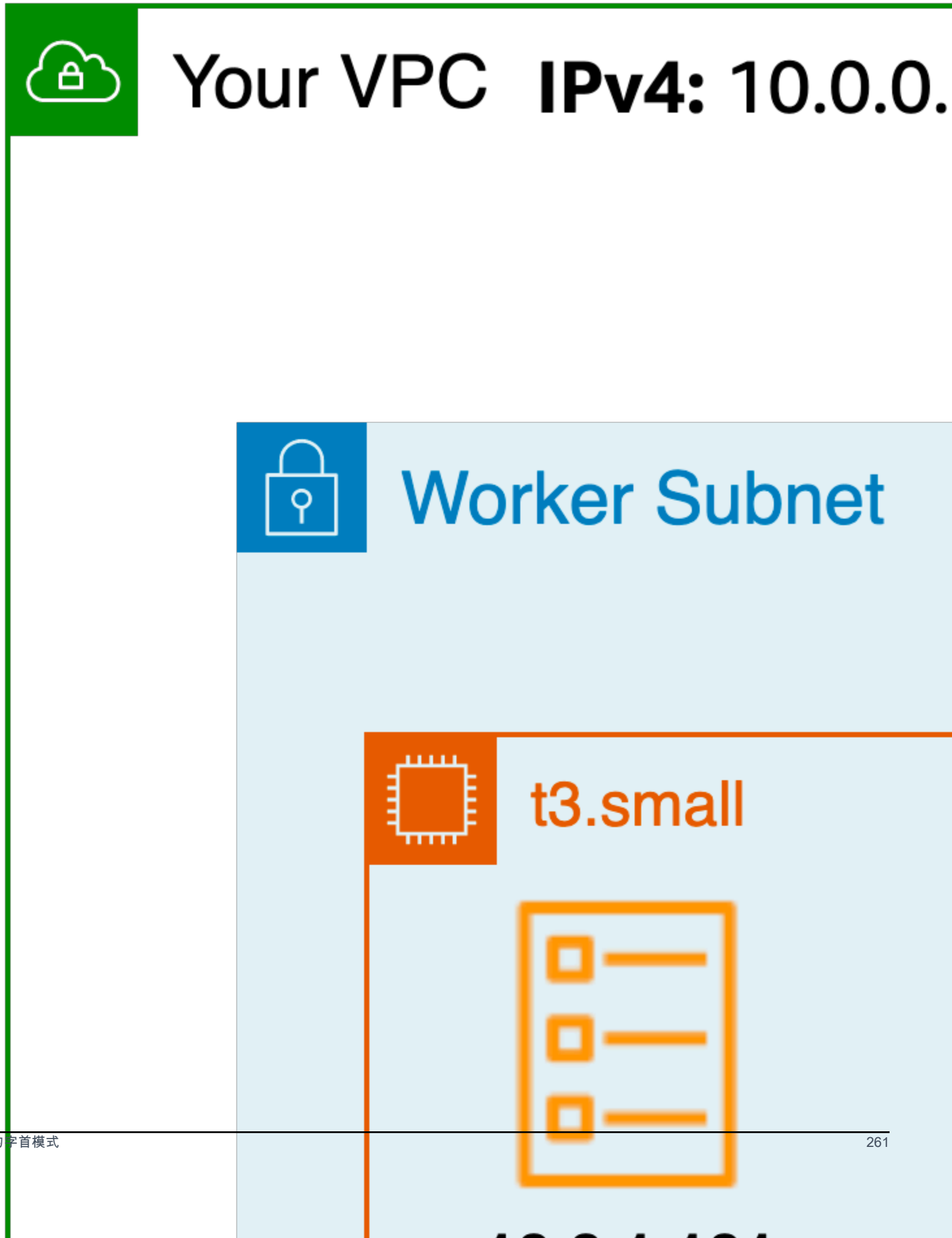
Linux 的字首模式

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

Amazon VPC CNI 會將網路字首指派給 Amazon EC2 網路介面，以增加節點可用的 IP 地址數量，並提高每個節點的 Pod 密度。您可以設定 Amazon VPC CNI 附加元件的 1.9.0 版或更新版本，以指派 IPv4 和 IPv6 CIDRs，而不是將個別次要 IP 地址指派給網路介面。

根據預設，IPv6 叢集上會啟用字首模式，並且是唯一支援的選項。VPC CNI 會將 /80 IPv6 字首指派給 ENI 上的插槽。如需詳細資訊，請參閱本指南的 IPv6 一節。

使用字首指派模式時，每個執行個體類型的彈性網路介面數量上限保持不變，但您現在可以將 Amazon VPC CNI 設定為指派 /28 (16 個 IP 地址） IPv4 地址字首，而不是將個別 IPv4 地址指派給網路介面上的插槽。當 `ENABLE_PREFIX_DELEGATION` 設為 true VPC CNI 時，會從指派給 ENI 的字首將 IP 地址配置到 Pod。請遵循 EKS 使用者指南中提及的指示來啟用字首 IP 模式。

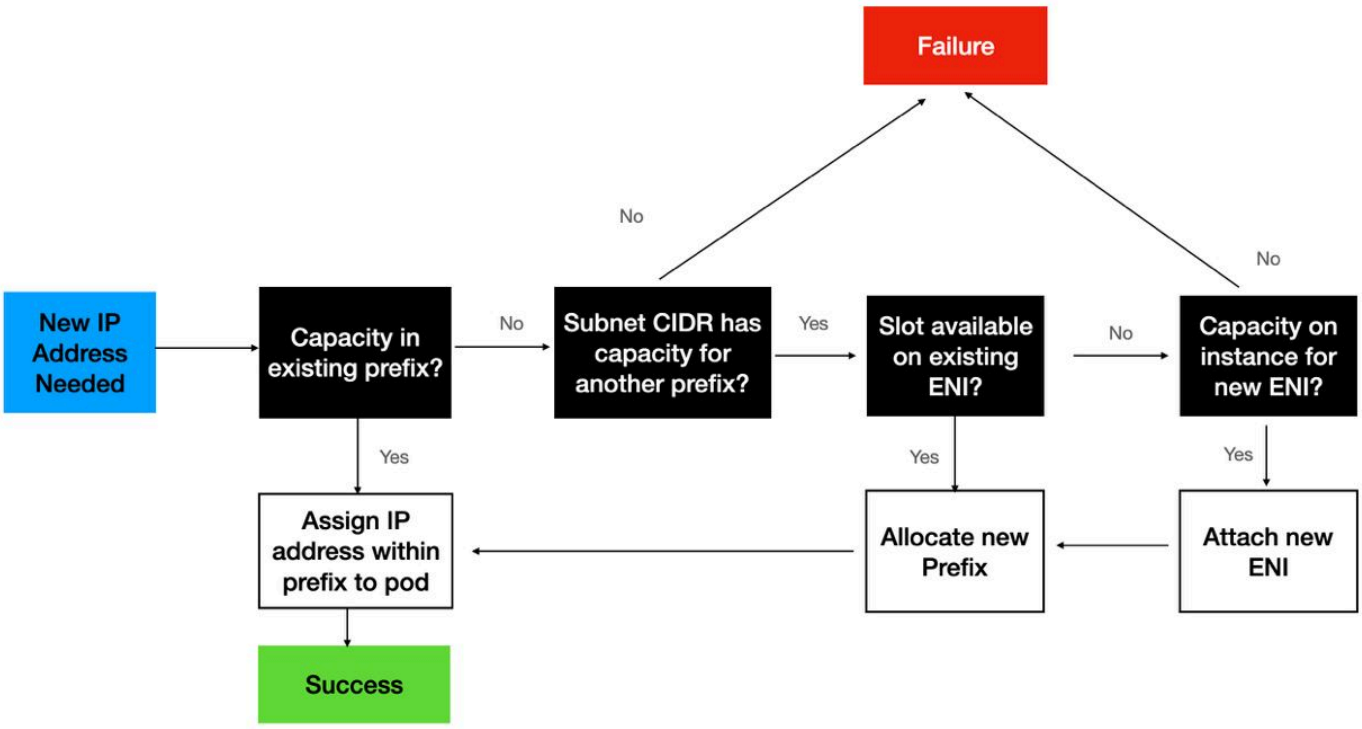

Your VPC IPv4: 10.0.0.
Worker Subnet
t3.small

您可以指派給網路介面的 IP 地址數量上限會因執行個體類型而異。指派給網路介面的每個字首都計為一個 IP 地址。例如，c5.large 執行個體每個網路介面的 IPv4 地址數量上限為 10 個。此執行個體的每個網路介面都具備主要 IPv4 地址。如果網路介面沒有次要 IPv4 地址，則最多可以為網路介面指派 9 個字首。為網路介面多指派一個 IPv4 地址，可為網路介面指派的字首數量便少一個。檢閱每個執行個體類型每個網路介面 IP 地址的 AWS EC2 文件，並將字首指派給網路介面。

在工作者節點初始化期間，VPC CNI 會將一或多個字首指派給主要 ENI。CNI 透過維護暖集區預先配置字首，以加快 Pod 啟動速度。可透過設定環境變數來控制暖集區中保留的字首數量。

- WARM_PREFIX_TARGET，要配置超過目前需求的字首數目。
- WARM_IP_TARGET，超過目前需求要配置的 IP 地址數量。
- MINIMUM_IP_TARGET，隨時可用的 IP 地址數量下限。
- WARM_IP_TARGET MINIMUM_IP_TARGET 如果設定 ， 和 會覆寫 WARM_PREFIX_TARGET。

由於現有 ENI 將請求更多 Pod 排程的其他字首。首先，VPC CNI 會嘗試將新的字首配置給現有的 ENI。如果 ENI 容量為 ，VPC CNI 會嘗試將新的 ENI 配置給節點。新的 ENIs將連接，直到達到 ENI 上限（由執行個體類型定義）。連接新的 ENI 時，ipamd 會配置維護 WARM_PREFIX_TARGET、WARM_IP_TARGET和 MINIMUM_IP_TARGET設定所需的一或多個字首。



建議

在 時使用字首模式

如果您在工作者節點上遇到 Pod 密度問題，請使用字首模式。為了避免 VPC CNI 錯誤，建議您先檢查子網路是否有 /28 字首地址的連續區塊，再遷移至字首模式。如需子網路保留詳細資訊，請參閱「使用子網路保留以避免子網路分段 (IPv4)」一節。

為了回溯相容性，最大 Pod 限制設定為支援次要 IP 模式。若要增加 Pod 密度，請將 `max-pods`值指定為 Kubelet 和 `--use-max-pods=false`，做為節點的使用者資料。如需詳細資訊，請參閱《Amazon EKS 使用者指南》中的 maxPods的判斷方式。如需使用者資料範例，請參閱 EKS 使用者指南。

```
./max-pods-calculator.sh --instance-type m5.large --cni-version ``1.9``.0 --cni-prefix-delegation-enabled
```

字首指派模式特別適用於 CNI 自訂聯網的使用者，其中主要 ENI 不用於 Pod。透過字首指派，即使沒有用於 Pod 的主要 ENI，您仍然可以在幾乎每個 Nitro 執行個體類型上連接更多 IPs。

在 時避免字首模式

如果您的子網路片段很大，且可用 IP 地址不足以建立 /28 字首，請避免使用字首模式。如果產生字首的子網路分段（具有分散次要 IP 地址的大量使用子網路），字首連接可能會失敗。建立新子網路並保留字首可避免此問題。

在字首模式中，指派給工作者節點的安全群組是由 Pod 共用。如果您有安全需求，可透過對共用運算資源執行具有不同網路安全要求的應用程式，來實現合規，請考慮使用 Pod 的安全群組。

在相同的節點群組中使用類似的執行個體類型

您的節點群組可能包含多種類型的執行個體。如果執行個體的 Pod 計數上限較低，則該值會套用至節點群組中的所有節點。請考慮在節點群組中使用類似的執行個體類型，以最大化節點使用量。如果您使用 Karpenter 進行自動節點擴展，我們建議您在佈建器 API 的要求部分中設定 node.kubernetes.io/instance-type。

> **Warning**
>
> 特定節點群組中所有節點的最大 Pod 計數是由節點群組中任何單一執行個體類型的最低最大 Pod 計數所定義。

設定 WARM_PREFIX_TARGET以保留 IPv4 地址

安裝資訊清單的預設值WARM_PREFIX_TARGET為 1。在大多數情況下， 的建議值 1 WARM_PREFIX_TARGET將提供快速 Pod 啟動時間的良好組合，同時將指派給執行個體的未使用 IP 地址降至最低。

如果您需要進一步保留每個節點的 IPv4 地址使用WARM_IP_TARGET和MINIMUM_IP_TARGET設定，這會在設定WARM_PREFIX_TARGET時覆寫。透過WARM_IP_TARGET將 設定為小於 16 的值，您可以防止 CNI 保持連接整個多餘的字首。

偏好配置新的字首而非連接新的 ENI

將額外的字首配置到現有的 ENI 比建立新的 ENI 並連接到執行個體更快的 EC2 API 操作。使用字首可改善效能，同時使用 IPv4 地址配置來保持原始狀態。連接字首通常會在一秒內完成，而連接新的 ENI 最多可能需要 10 秒。對於大多數使用案例，在字首模式下執行時，CNI 只需要每個工作者節點單一 ENI。如果您可以負擔（在最壞的情況下）每個節點最多 15 IPs，我們強烈建議使用較新的字首指派聯網模式，並實現其隨附的效能和效率增益。

使用子網路預留來避免子網路分段 (IPv4)

當 EC2 將 /28 IPv4 字首配置給 ENI 時，它必須是來自子網路的連續 IP 地址區塊。如果產生字首的子網路是分段的（具有分散次要 IP 地址的高度使用子網路），字首連接可能會失敗，而且您會在 VPC CNI 日誌中看到下列錯誤訊息：

```
failed to allocate a private IP/Prefix address: InsufficientCidrBlocks: There are not
 enough free cidr blocks in the specified subnet to satisfy the request.
```

為了避免分段並有足夠的連續空間來建立字首，您可以使用 VPC 子網路 CIDR 保留來保留子網路內的 IP 空間，以供字首使用。建立保留後，VPC CNI 外掛程式會呼叫 EC2 APIs，以指派從保留空間自動配置的字首。

建議建立新的子網路、為字首預留空間，並為在該子網路中執行的工作者節點使用 VPC CNI 啟用字首指派。如果新的子網路專用於已啟用 VPC CNI 字首指派的 EKS 叢集中執行的 Pod，則可以略過字首保留步驟。

避免降級 VPC CNI

字首模式適用於 VPC CNI 1.9.0 版及更新版本。啟用字首模式並將字首指派給 ENIs版本。如果您決定降級 VPC CNI，您必須刪除並重新建立節點。

在轉換為字首委派期間取代所有節點

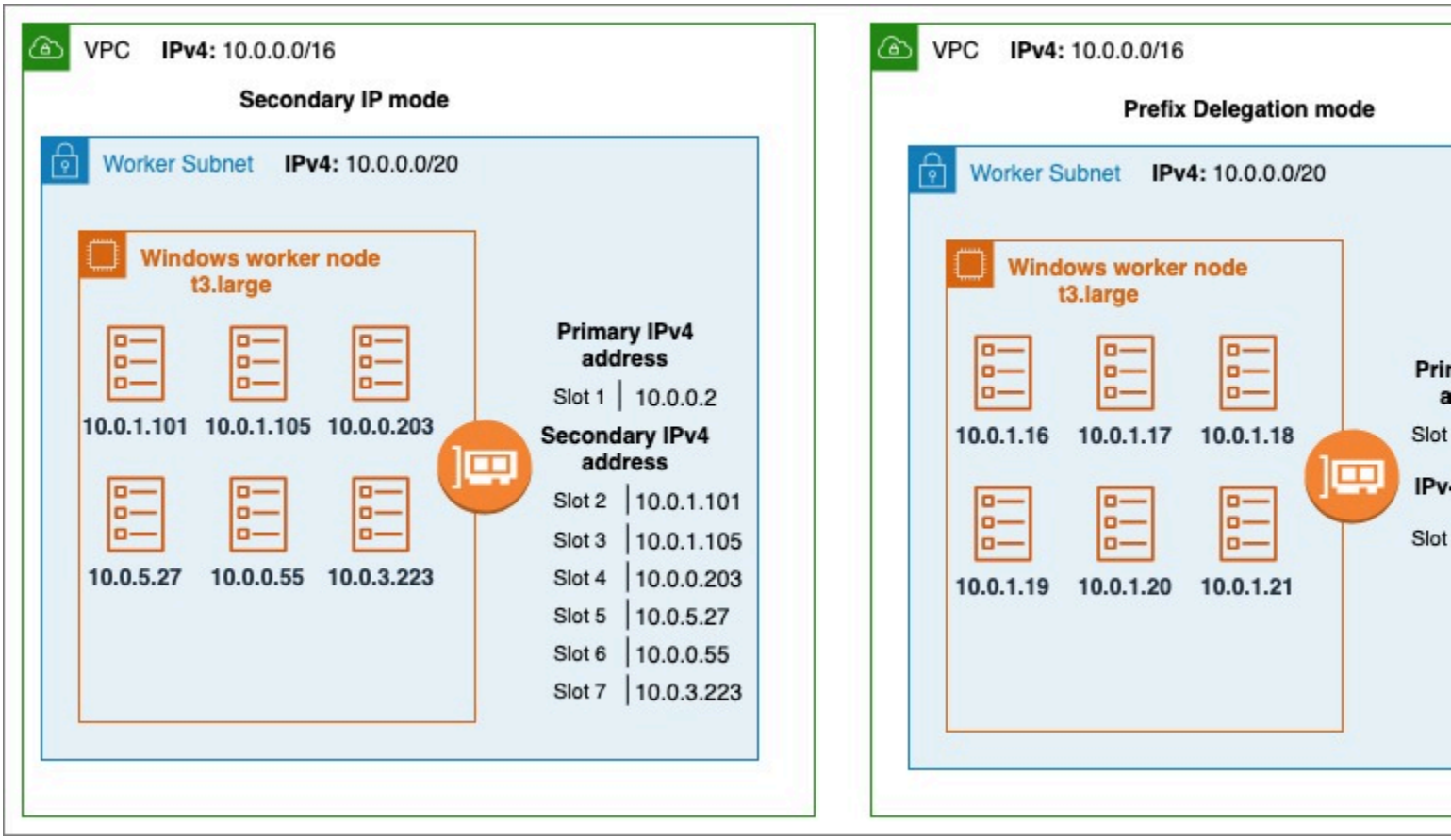
強烈建議您建立新的節點群組，以增加可用 IP 地址的數量，而不是輪流取代現有的工作者節點。封鎖並耗盡所有現有的節點，以安全地移出所有現有的 Pod。為了防止服務中斷，我們建議您在關鍵工作負載的生產叢集上實作 Pod 中斷預算。新節點上的 Pod 將從指派給 ENI 的字首指派 IP。確認 Pod 正在執行後，您可以刪除舊節點和節點群組。如果您使用的是受管節點群組，請依照此處提及的步驟安全地刪除節點群組。

Windows 的字首模式

在 Amazon EKS 中，在 Windows 主機上執行的每個 Pod 預設都會由 VPC 資源控制器指派次要 IP 地址。此 IP 地址是從主機子網路配置的 VPC 可路由地址。在 Linux 上，連接至執行個體的每個 ENI 都有多個插槽，可由次要 IP 地址或 /28 CIDR（字首）填入。不過，Windows 主機僅支援單一 ENI 及其可用插槽。只使用次要 IP 地址可以人工限制您可以在 Windows 主機上執行的 Pod 數量，即使有大量的 IP 地址可供指派。

為了增加 Windows 主機上的 Pod 密度，特別是在使用較小的執行個體類型時，您可以為 Windows 節點啟用字首委派。啟用字首委派時，會將 /28 IPv4 字首指派給 ENI 插槽，而不是次要 IP 地址。前綴委派可以透過將 `enable-windows-prefix-delegation: "true"`項目新增至`amazon-vpc-cni`組態映射來啟用。這與您需要設定`enable-windows-ipam: "true"`項目以啟用 Windows 支援的組態映射相同。

請遵循 EKS 使用者指南中提及的指示，為 Windows 節點啟用字首委派模式。



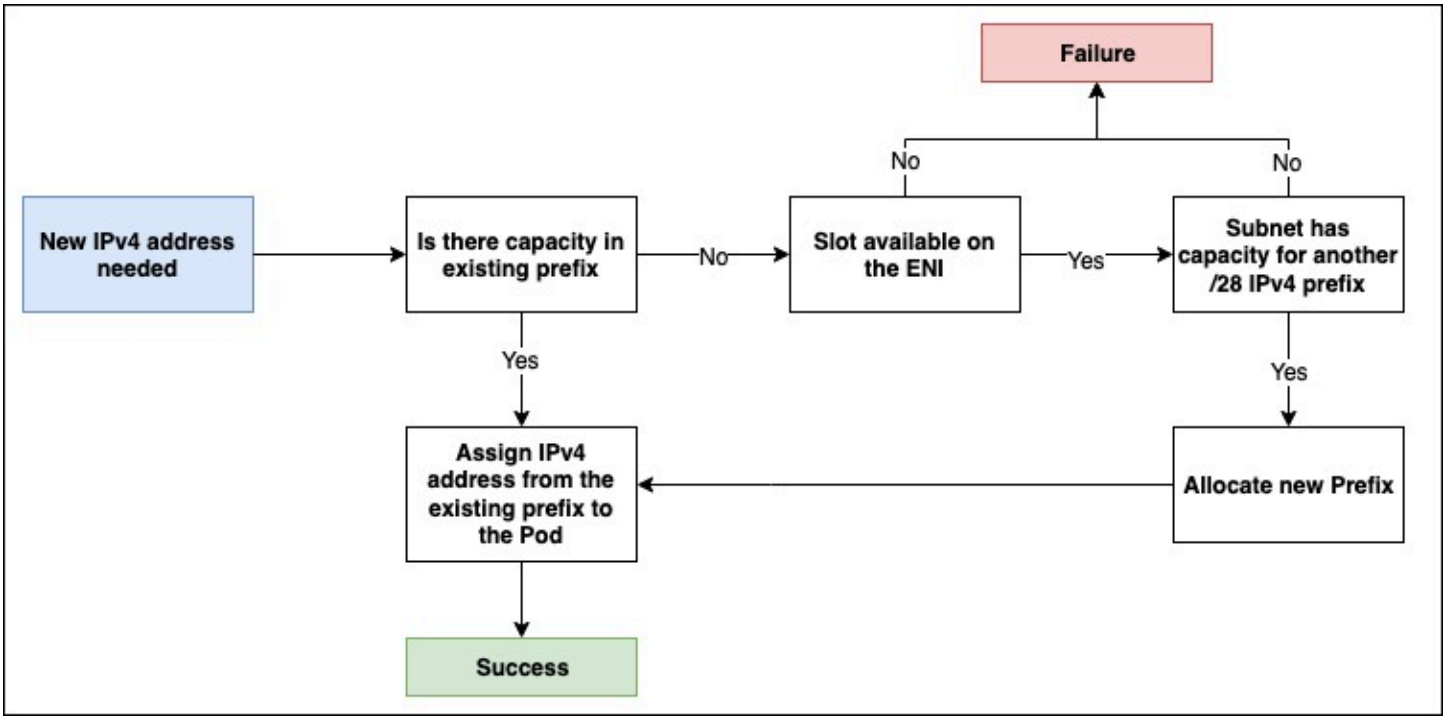
圖：次要 IP 模式與字首委派模式的比較

您可以指派給網路介面的 IP 地址數目上限取決於執行個體類型及其大小。指派給網路界面的每個字首都會使用可用的插槽。例如，`c5.large`執行個體每個網路界面有 個10插槽的限制。網路介面上的第一個插槽一律會由介面的主要 IP 地址耗用，讓您擁有 9 個字首和/或次要 IP 地址的插槽。如果已指派這些插槽字首，節點可以支援 (9 * 16) 144 個 IP 地址，如果已指派次要 IP 地址，則只能支援 9 個 IP 地址。如需詳細資訊，請參閱每個執行個體類型每個網路介面 IP 地址的文件，以及將字首指派給網路介面。

在工作者節點初始化期間，VPC Resource Controller 會維護 IP 地址的暖集區，將一或多個字首指派給主要 ENI，以加快 Pod 啟動速度。您可以在`amazon-vpc-cni`組態映射中設定下列組態參數，以控制暖集區中保留的字首數量。

- `warm-prefix-target`，要配置的字首數目超過目前需求。
- `warm-ip-target`，超過目前需求要配置的 IP 地址數量。
- `minimum-ip-target`，隨時可用的 IP 地址數量下限。
- `warm-ip-target` 和/或`minimum-ip-target`如果設定 將覆寫 `warm-prefix-target`。

當節點上排程了更多 Pod 時，將為現有的 ENI 請求額外的字首。在節點上排程 Pod 時，VPC 資源控制器會先嘗試從節點上現有的字首指派 IPv4 地址。如果無法這麼做，只要子網路具有所需的容量，就會請求新的 IPv4 字首。



圖：指派 IPv4 地址至 Pod 時的工作流程

建議

在 時使用字首委派

如果您在工作者節點上遇到 Pod 密度問題，請使用字首委派。為了避免錯誤，建議您在遷移至字首模式之前，先檢查子網路是否有 /28 字首的連續地址區塊。如需子網路保留詳細資訊，請參閱「使用子網路保留來避免子網路分段 (IPv4)」一節。

根據預設，Windows 節點`max-pods`上的 會設定為 `110`。對於絕大多數的執行個體類型，這應該已足夠。如果您想要增加或減少此限制，請在使用者資料中的引導命令中新增下列項目：

```
-KubeletExtraArgs '--max-pods=example-value'
```

如需 Windows 節點引導組態參數的詳細資訊，請參閱此處的文件。

在 時避免字首委派

如果您的子網路片段很大，且可用 IP 地址不足以建立 /28 字首，請避免使用字首模式。如果產生字首的子網路分段（具有分散次要 IP 地址的大量使用子網路），字首連接可能會失敗。建立新子網路並保留字首可避免此問題。

設定字首委派的參數以保留 IPv4 地址

`warm-prefix-target`、 `warm-ip-target`和 `minimum-ip-target`可用來使用字首微調預先擴展和動態擴展的行為。根據預設，會使用下列值：

```
warm-ip-target: "1"
minimum-ip-target: "3"
```

透過微調這些組態參數，您可以實現最佳平衡，以保留 IP 地址並確保因指派 IP 地址而降低 Pod 延遲。如需這些組態參數的詳細資訊，請參閱此處的文件。

使用子網路預留來避免子網路分段 (IPv4)

當 EC2 將 /28 IPv4 字首配置給 ENI 時，它必須是來自子網路的連續 IP 地址區塊。如果產生字首的子網路是分段的（具有分散次要 IP 地址的高度使用子網路），字首連接可能會失敗，而且您會看到下列節點事件：

```
InsufficientCidrBlocks: The specified subnet does not have enough free cidr blocks to
 satisfy the request
```

為了避免分段並有足夠的連續空間來建立字首，請使用 VPC 子網路 CIDR 保留來保留子網路內的 IP 空間，以供字首使用。建立保留後，不會將保留區塊中的 IP 地址指派給其他資源。如此一來，VPC 資源控制器就能在對節點 ENI 的指派呼叫期間取得可用的字首。

建議建立新的子網路、為字首預留空間，並為在該子網路中執行的工作者節點啟用字首指派。如果新的子網路專用於在啟用字首委派的 EKS 叢集中執行的 Pod，則可以略過字首保留步驟。

從次要 IP 模式遷移至字首委派模式時取代所有節點，反之亦然

強烈建議您建立新的節點群組，以增加可用 IP 地址的數量，而不是輪流取代現有的工作者節點。

使用自我管理節點群組時，轉換的步驟如下：

- 增加叢集中的容量，讓新節點能夠容納您的工作負載

- 啟用/停用 Windows 的字首委派功能
- 封鎖並耗盡所有現有的節點，以安全地移出所有現有的 Pod。為了防止服務中斷，我們建議您在關鍵工作負載的生產叢集上實作 Pod 中斷預算。
- 確認 Pod 正在執行後，您可以刪除舊節點和節點群組。新節點上的 Pod 將從指派給節點 ENI 的字首指派 IPv4 地址。

使用受管節點群組時，轉換的步驟如下：

- 啟用/停用 Windows 的字首委派功能
- 使用此處提及的步驟更新節點群組。這會執行與上述類似的步驟，但由 EKS 管理。

> **Warning**
>
> 在相同模式的節點上執行所有 Pod

對於 Windows，我們建議您避免同時在次要 IP 模式和字首委派模式下執行 Pod。當您從次要 IP 模式遷移至字首委派模式，反之亦然執行 Windows 工作負載時，可能會發生這種情況。

雖然這不會影響您執行中的 Pod，但節點的 IP 地址容量可能不一致。例如，假設 t3.xlarge 節點有 14 個用於次要 IPv4 地址的插槽。如果您執行 10 個 Pod，則次要 IP 地址會耗用 ENI 上的 10 個插槽。啟用字首委派後，公告至 kube-api 伺服器的容量為（每個字首 14 個插槽 * 16 個 ip 地址）244，但目前實際容量為（每個字首 4 個剩餘插槽 * 16 個地址）64。如果您執行的 Pod 數量超過可供指派的 IP 地址，則公告的容量與實際容量（剩餘插槽）之間的這種不一致可能會導致問題。

也就是說，您可以使用上述的遷移策略，將 Pod 從次要 IP 地址安全地轉換為從字首取得的地址。在模式之間切換時，Pod 會繼續正常執行，並且：

- 從次要 IP 模式切換到字首委派模式時，不會釋出指派給執行中 Pod 的次要 IP 地址。字首將指派給可用插槽。一旦 Pod 終止，將會釋出其正在使用的次要 IP 和插槽。
- 從字首委派模式切換到次要 IP 模式時，當範圍內的所有 IPs 不再配置給 Pod 時，將會釋放字首。如果任何來自 字首的 IP 指派給 Pod，則該字首會保留到 Pod 終止為止。

使用字首委派對問題進行偵錯

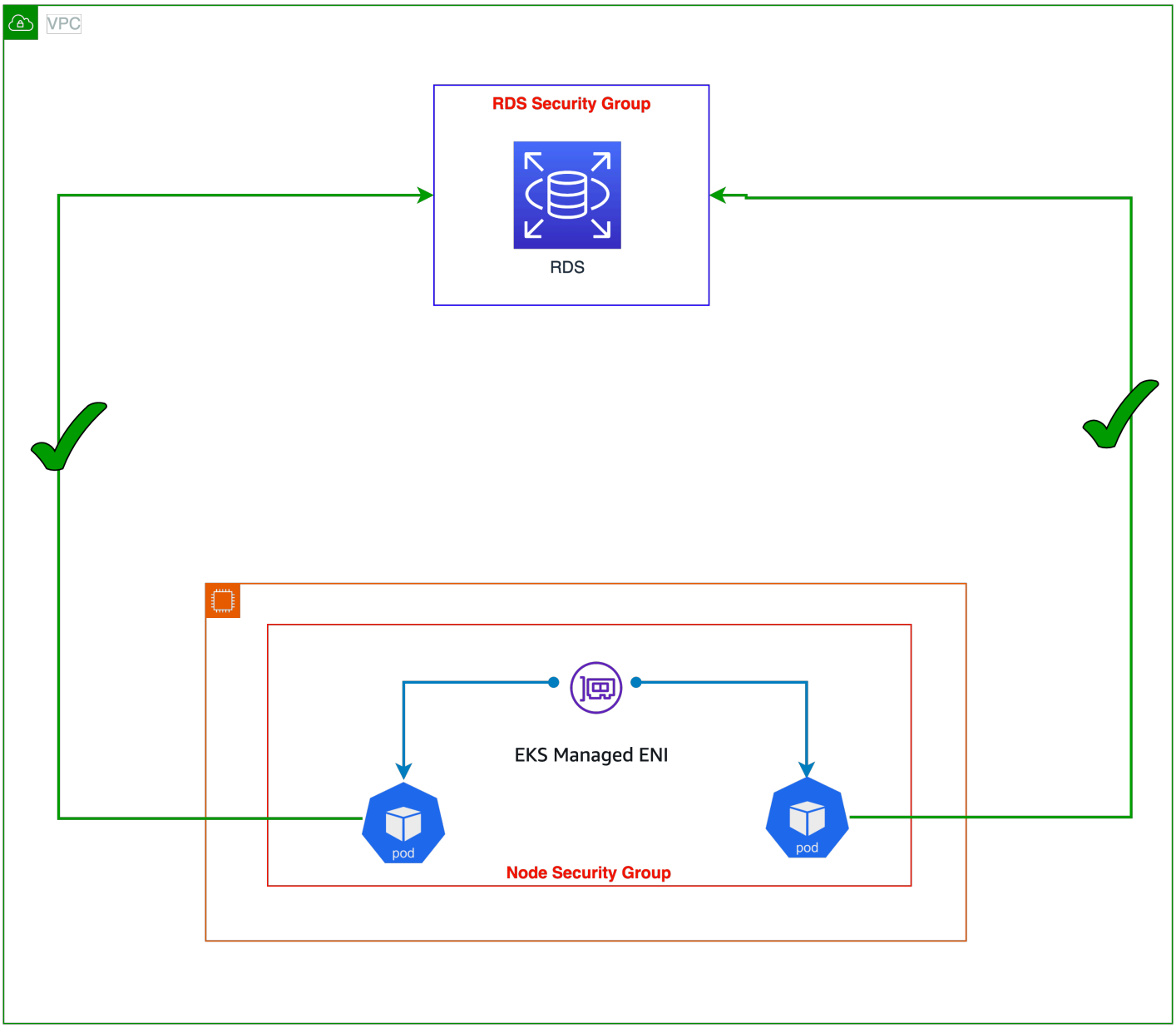
您可以使用此處的除錯指南https://github.com/aws/amazon-vpc-resource-controller-k8s/blob/master/docs/troubleshooting.md，深入了解您在 Windows 上字首委派所面臨的問題。

每個 Pod 的安全群組

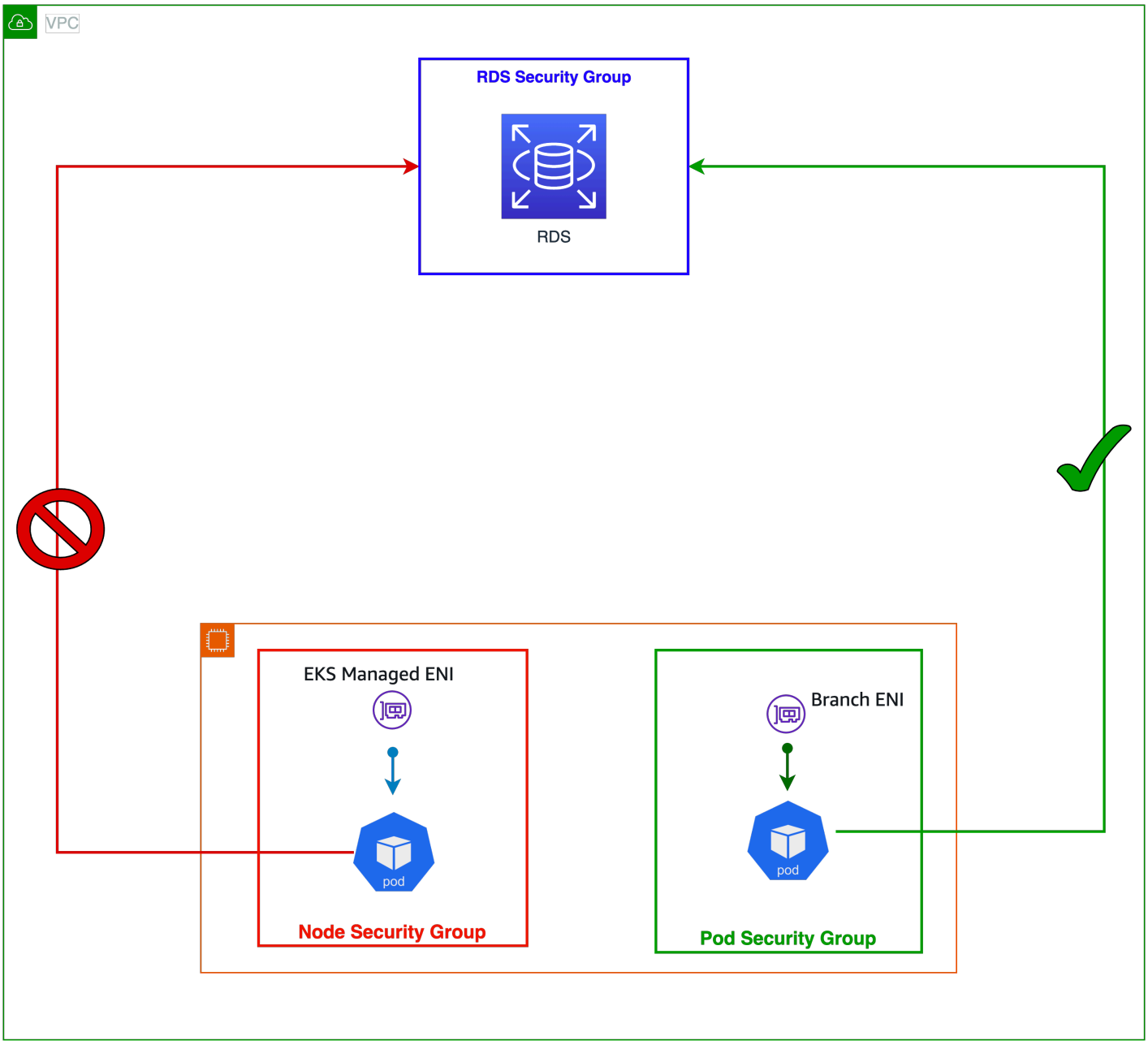
> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

AWS 安全群組可做為 EC2 執行個體的虛擬防火牆，以控制傳入和傳出流量。根據預設，Amazon VPC CNI 將使用與節點上主要 ENI 相關聯的安全群組。更具體地說，與執行個體相關聯的每個 ENI 都會有相同的 EC2 安全群組。因此，節點上的每個 Pod 都會與其執行節點共用相同的安全群組。

如下圖所示，在工作者節點上執行的所有應用程式 Pod 都可以存取 RDS 資料庫服務（考量 RDS 傳入允許節點安全群組）。安全群組太粗精細，因為它們適用於節點上執行的所有 Pod。Pod 的安全群組為工作負載提供網路分段，這是良好深度防禦策略的重要部分。

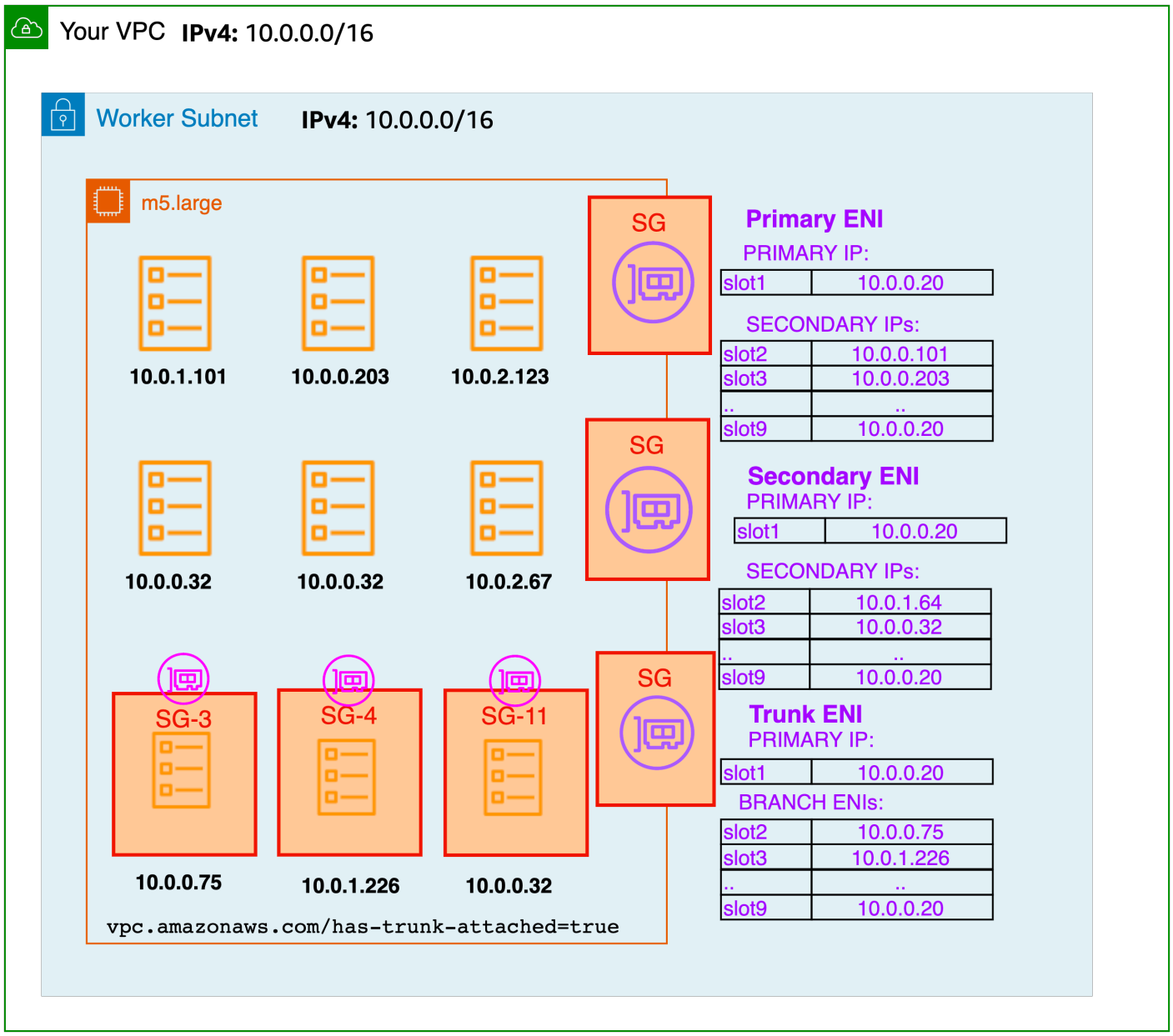


透過 Pod 的安全群組，您可以執行對共用運算資源具有不同網路安全需求的應用程式，以改善運算效率。多種類型的安全規則，例如 Pod-to-Pod 和 Pod-to-External AWS 服務，可以使用 EC2 安全群組在單一位置定義，並使用 Kubernetes 原生 APIs 套用至工作負載。下圖顯示 Pod 層級套用的安全群組，以及它們如何簡化您的應用程式部署和節點架構。Pod 現在可存取 Amazon RDS 資料庫。



您可以透過設定 VPC CNI `ENABLE_POD_ENI=true`來啟用 Pod 的安全群組。啟用後，在控制平面上執行的 VPC 資源控制器（由 EKS 管理）會建立並連接名為 "`aws-k8s-trunk-eni"` 的幹線界面至節點。幹線界面可做為連接至執行個體的標準網路界面。若要管理幹線介面，您必須將 `AmazonEKSVPCResourceController`受管政策新增至 Amazon EKS 叢集隨附的叢集角色。

控制器也會建立名為 "aws-k8s-branch-eni" 的分支界面，並將其與幹線界面建立關聯。Pod 會使用 SecurityGroupPolicy 自訂資源指派安全群組，並與分支介面相關聯。由於安全群組是以網路介面指定，我們現在能夠在這些額外的網路介面上排程需要特定安全群組的 Pod。檢閱 Pod 安全群組的 EKS 使用者指南章節，包括部署先決條件。



分支介面容量會附加至次要 IP 地址的現有執行個體類型限制。使用安全群組的 Pod 不會計入 max-Pod 公式中，當您將安全群組用於 Pod 時，您需要考慮提高 max-Pod 值，或者執行的 Pod 數量比節點實際支援的 Pod 少。

m5.large 最多可指派 9 個分支網路介面，以及最多 27 個次要 IP 地址給其標準網路介面。如以下範例所示，m5.large 的預設最大 Pod 為 29，EKS 會將使用安全群組的 Pod 計入最大 Pod。如需如何變更節點最大 Pod 的說明，請參閱 EKS 使用者指南。

當 Pod 的安全群組與自訂聯網搭配使用時，會使用 Pod 安全群組中定義的安全群組，而不是 ENIConfig 中指定的安全群組。因此，啟用自訂聯網時，請仔細評估安全群組排序，同時使用每個 Pod 的安全群組。

建議

停用 TCP Early Demux for Liveness 探查

如果您是使用即時或整備探查，您還需要停用 TCP 早期 demux，以便 kubelet 可以透過 TCP 連接到分支網路介面上的 Pod。這只有在嚴格模式下才需要。若要執行此操作，請執行下列命令：

```
kubectl edit daemonset aws-node -n kube-system
```

在 `initContainer`區段下，將 的值變更為 `DISABLE_TCP_EARLY_DEMUX` `true`.

使用安全群組 Pod 來利用現有的 AWS 組態投資。

安全群組可讓您更輕鬆地限制對 VPC 資源的網路存取，例如 RDS 資料庫或 EC2 執行個體。每個 Pod 的安全群組的一個明顯優勢是有機會重複使用現有的 AWS 安全群組資源。如果您使用安全群組做為網路防火牆來限制對 AWS 服務的存取，建議您使用分支 ENIs 將安全群組套用至 Pod。如果您要將應用程式從 EC2 執行個體轉移到 EKS，並限制存取具有安全群組的其他 AWS 服務，請考慮使用 Pod 的安全群組。

設定 Pod 安全群組強制執行模式

Amazon VPC CNI 外掛程式 1.11 版新增了名為 `POD_SECURITY_GROUP_ENFORCING_MODE`(「強制執行模式」) 的新設定。強制執行模式會控制哪些安全群組適用於 Pod，以及是否啟用來源 NAT。您可以指定強制執行模式為嚴格或標準。嚴格是預設值，反映將 `ENABLE_POD_ENI`設為 的 VPC CNI 先前行為`true`。

在嚴格模式中，只會強制執行分支 ENI 安全群組。來源 NAT 也會停用。

在標準模式中，會套用與主要 ENI 和分支 ENI （與 Pod 關聯） 相關聯的安全群組。網路流量必須符合兩個安全群組。

> **Warning**
>
> 任何模式變更都只會影響新啟動的 Pod。現有的 Pod 將使用建立 Pod 時所設定的模式。如果客戶想要變更流量行為，則需要回收具有安全群組的現有 Pod。

強制執行模式：使用嚴格模式隔離 Pod 和節點流量：

根據預設，Pod 的安全群組會設定為「嚴格模式」。如果您必須完全將 Pod 流量與節點的其餘流量分開，請使用此設定。在嚴格模式下，來源 NAT 會關閉，以便可以使用分支 ENI 傳出安全群組。

> **Warning**
>
> 啟用嚴格模式時，來自 Pod 的所有傳出流量都會離開節點並進入 VPC 網路。相同節點上 Pod 之間的流量會經過 VPC。這會增加 VPC 流量並限制節點型功能。嚴格模式不支援 NodeLocal DNSCache。

強制執行模式：在下列情況下使用標準模式

Pod 中容器可見的用戶端來源 IP

如果您需要讓 Pod 中的容器可看見用戶端來源 IP，請考慮`POD_SECURITY_GROUP_ENFORCING_MODE`將 設定為 `standard`。Kubernetes 服務支援 externalTrafficPolicy=local，以支援保留用戶端來源 IP（預設類型叢集）。您現在可以使用執行個體目標執行 NodePort 和 LoadBalancer 類型的 Kubernetes 服務，並在標準模式中將 externalTrafficPolicy 設定為 Local。 會`Local`保留用戶端來源 IP，並避免 LoadBalancer 和 NodePort 類型的服務的第二個跳轉。

部署 NodeLocal DNSCache

使用 Pod 的安全群組時，請設定標準模式以支援使用 NodeLocal DNSCache 的 Pod。NodeLocal DNSCache 透過在叢集節點上執行 DNS 快取代理程式作為 DaemonSet 來改善叢集 DNS 效能。這將有助於具有最高 DNS QPS 需求的 Pod 查詢具有本機快取的本機 kube-dns/CoreDNS，這將改善延遲。

NodeLocal DNSCache 在嚴格模式下不受支援，因為即使節點的所有網路流量都會進入 VPC。

支援 Kubernetes 網路政策

我們建議您在搭配具有相關聯安全群組的 Pod 使用網路政策時，使用標準強制執行模式。

我們強烈建議為 Pod 使用安全群組，以限制網路層級存取不屬於叢集的 AWS 服務。考慮網路政策來限制叢集內 Pod 之間的網路流量，通常稱為東部/西部流量。

識別每個 Pod 與安全群組的不相容

Windows 型和非 nitro 執行個體不支援 Pod 的安全群組。若要搭配 Pod 使用安全群組，執行個體必須以 isTrunkingEnabled 標記。如果您的 Pod 不依賴 VPC 內外的任何 AWS 服務，請使用網路政策來管理 Pod 之間的存取，而不是安全群組。

使用每個 Pod 的安全群組，有效率地控制 AWS Services 的流量

如果在 EKS 叢集中執行的應用程式必須與 VPC 中的其他資源通訊，例如 RDS 資料庫，請考慮使用 SGs進行 Pod。雖然有政策引擎可讓您指定 CIDR 或 DNS 名稱，但與端點位於 VPC 內的 AWS 服務通訊時，它們是較不理想的選擇。

相反地，Kubernetes 網路政策提供控制叢集內外輸入和輸出流量的機制。如果您的應用程式對其他 AWS 服務的相依性有限，應考慮使用 Kubernetes 網路政策。您可以設定網路政策，根據 CIDR 範圍指定輸出規則，以限制對 AWS 服務的存取，而不是 SGs 等 AWS 原生語意。您可以使用 Kubernetes 網路政策來控制 Pod 之間的網路流量（通常稱為東部/西部流量），以及 Pod 與外部服務之間的網路流量。Kubernetes 網路政策會在 OSI 第 3 級和第 4 級實作。

Amazon EKS 可讓您使用 Calico 和 Cilium 等網路政策引擎。根據預設，不會安裝網路政策引擎。如需如何設定的指示，請參閱各自的安裝指南。如需如何使用網路政策的詳細資訊，請參閱 EKS 安全最佳實務。DNS 主機名稱功能適用於網路政策引擎的企業版本，這可能有助於控制 Kubernetes Services/Pod 與在 AWS 外部執行的資源之間的流量。此外，您也可以考慮對預設不支援安全群組的 AWS 服務的 DNS 主機名稱支援。

標記單一安全群組以使用 AWS Loadbalancer 控制器

當許多安全群組配置到 Pod 時，Amazon EKS 建議標記具有`kubernetes.io/cluster/$name`共用或擁有的單一安全群組。標籤允許 AWS Loadbalancer 控制器更新安全群組的規則，將流量路由到 Pod。如果只將一個安全群組提供給 Pod，則標籤的指派是選擇性的。安全群組中設定的許可是累加的，因此標記單一安全群組就足以讓負載平衡器控制器尋找和協調規則。它也有助於遵守安全群組定義的預設配額。

設定傳出流量的 NAT

來源 NAT 會針對來自指派安全群組之 Pod 的傳出流量停用。對於使用安全群組的 Pod，這些安全群組需要存取使用 NAT 閘道或執行個體設定的私有子網路上的網際網路啟動工作者節點，並在 CNI 中啟用外部 SNAT。

```
kubectl set env daemonset -n kube-system aws-node AWS_VPC_K8S_CNI_EXTERNALSNAT=true
```

使用安全群組將 Pod 部署至私有子網路

指派給安全群組的 Pod 必須在部署到私有子網路的節點上執行。請注意，已將指派安全群組部署到公有子網路的 Pod 將無法存取網際網路。

在 Pod 規格檔案中驗證 terminationGracePeriodSeconds

請確定 Pod 規格檔案中`terminationGracePeriodSeconds`的 不是零 （預設 30 秒）。這對於 Amazon VPC CNI 從工作者節點刪除 Pod 網路至關重要。設為零時，CNI 外掛程式不會從主機移除 Pod 網路，而且分支 ENI 不會有效清除。

將 Pod 的安全群組與 Fargate 搭配使用

在 Fargate 上執行之 Pod 的安全群組的運作方式與在 EC2 工作者節點上執行的 Pod 非常類似。例如，您必須先建立安全群組，才能在與 Fargate Pod 建立關聯的 SecurityGroupPolicy 中參考安全群組。根據預設，當您未明確將 SecurityGroupPolicy 指派給 Fargate Pod 時，叢集安全群組會連接至所有 Fargate Pod。為了簡單起見，您可能想要將叢集安全群組新增至 Fagate Pod 的 SecurityGroupPolicy，否則您必須將最低安全群組規則新增至安全群組。您可以使用 describe-cluster API 尋找叢集安全群組。

```
 aws eks describe-cluster --name CLUSTER_NAME --query
 'cluster.resourcesVpcConfig.clusterSecurityGroupId'
```

```
cat >my-fargate-sg-policy.yaml <<EOF
apiVersion: vpcresources.k8s.aws/v1beta1
kind: SecurityGroupPolicy
metadata:
  name: my-fargate-sg-policy
  namespace: my-fargate-namespace
spec:
  podSelector:
    matchLabels:
      role: my-fargate-role
  securityGroups:
    groupIds:
      - cluster_security_group_id
      - my_fargate_pod_security_group_id
EOF
```

此處列出最低安全群組規則。這些規則允許 Fargate Pods 與叢集內服務通訊，例如 kube-apiserver、kubelet 和 CoreDNS。您也需要新增規則，以允許進出 Fargate Pod 的傳入和傳出連

線。這可讓您的 Pod 與 VPC 中的其他 Pod 或資源通訊。此外，您必須包含 Fargate 規則，才能從 Amazon ECR 或其他容器登錄檔提取容器映像，例如 DockerHub。如需詳細資訊，請參閱 AWS 一般參考中的 AWS IP 地址範圍。

您可以使用下列命令來尋找套用至 Fargate Pod 的安全群組。

```
kubectl get pod FARGATE_POD -o jsonpath='{.metadata.annotations.fargate\.amazonaws
\.com/pod-sg}{"\n"}'
```

從上述命令記下 eniId。

```
aws ec2 describe-network-interfaces --network-interface-ids ENI_ID --query
 'NetworkInterfaces[*].Groups[*]'
```

必須刪除並重新建立現有的 Fargate Pod，才能套用新的安全群組。例如，下列命令會啟動 example-app 的部署。若要更新特定 Pod，您可以在下列命令中變更命名空間和部署名稱。

```
kubectl rollout restart -n example-ns deployment example-pod
```

負載平衡

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

負載平衡器會接收傳入流量，並將其分佈到 EKS 叢集中託管的預期應用程式目標。這可改善應用程式的彈性。在 EKS 叢集中部署時，AWS Load Balancer 控制器會為該叢集建立和管理 AWS Elastic Load Balancer。建立類型 LoadBalancer 的 Kubernetes 服務時，AWS Load Balancer 控制器會建立 Network Load Balancer (NLB)，以平衡 OSI 模型第 4 層接收的流量。建立 Kubernetes 輸入物件時，AWS Load Balancer 控制器會建立 Application Load Balancer (ALB)，以平衡 OSI 模型第 7 層的流量。

選擇Load Balancer類型

AWS Elastic Load Balancing (ELB) 產品組合支援下列負載平衡器：Application Load Balancer (ALB)、Network Load Balancer (NLB)、Gateway Load Balancer (GWLB) 和 Classic Load Balancer (CLB)。此最佳實務區段將著重於 ALB 和 NLB，這是與 EKS 叢集最相關的兩個。

選擇負載平衡器類型的主要考量是工作負載需求。

如需更多詳細資訊並做為所有 AWS Load Balancer 的參考，請參閱產品比較

如果您的工作負載是 HTTP/HTTPS，請選擇 Application Load Balancer (ALB)

如果工作負載需要在 OSI 模型的第 7 層進行負載平衡，AWS Load Balancer 控制器可用於佈建 ALB；我們涵蓋下一節的佈建。ALB 由先前提到的輸入資源控制和設定，並將 HTTP 或 HTTPS 流量路由到叢集中的不同 Pod。ALB 可讓客戶靈活地變更應用程式流量路由演算法；預設路由演算法是循環路由演算法，具有最不未完成的請求路由演算法也是替代方案。

如果您的工作負載是 TCP，或者您的工作負載需要用戶端的來源 IP 保留，請選擇 Network Load Balancer (NLB)

Network Load Balancer 函數位於開放式系統互連 (OSI) 模型的第四個層 （ 傳輸 ）。它適用於 TCP 和 UDP 型工作負載。根據預設，Network Load Balancer 在將流量呈現至 Pod 時，也會保留用戶端地址的來源 IP。

如果您的工作負載無法使用 DNS，請選擇 Network Load Balancer (NLB)

使用 NLB 的另一個主要原因是，如果您的用戶端無法使用 DNS。在這種情況下，NLB 可能更適合您的工作負載，因為 Network Load Balancer IPs 是靜態的。雖然建議用戶端在連線至負載平衡器時將網域名稱解析為 IP 地址時使用 DNS，但如果用戶端的應用程式不支援 DNS 解析且只接受硬式編碼 IPs，則 NLB 更適合，因為 IPs是靜態的，並且在 NLB 的生命週期內保持不變。

佈建負載平衡器

在判斷最適合工作負載的Load Balancer之後，客戶有許多選項可以佈建負載平衡器。

部署 AWS Load Balancer 控制器來佈建Load Balancer

在 EKS 叢集中佈建負載平衡器有兩種關鍵方法。

- 在 AWS 雲端提供者中利用服務控制器 （ 舊版 ）
- 利用 AWS Load Balancer 控制器 （ 建議 ）

根據預設，Kubernetes Service Controller 也稱為樹狀內控制器，會協調 LoadBalancer 類型的 Kubernetes Service 資源。此控制器內建在 AWS Cloud Provider 元件中，可做為 Kubernetes Cloud Controller Manager。

佈建 Elastic Load Balancer 的組態是由必須新增至 Kubernetes Service 資訊清單的註釋所控制。服務控制器和 AWS Load Balancer 控制器所使用的註釋不同。

服務控制器是舊版的，目前僅收到重要的錯誤修正。當您建立 LoadBalancer 類型的 Kubernetes 服務時，服務控制器預設會建立 AWS CLB，但如果您使用正確的註釋，也可以建立 AWS NLB。值得注意的是，Service Controller 不支援 Kubernetes Ingress 資源，也不支援 IPv6。

我們建議您在 EKS 叢集中使用 AWS Load Balancer 控制器，以協調 Kubernetes Service 和輸入資源。您必須在 Kubernetes 服務或輸入資訊清單中使用正確的註釋，以便 AWS Load Balancer 控制器擁有對帳程序。（而不是服務控制器）

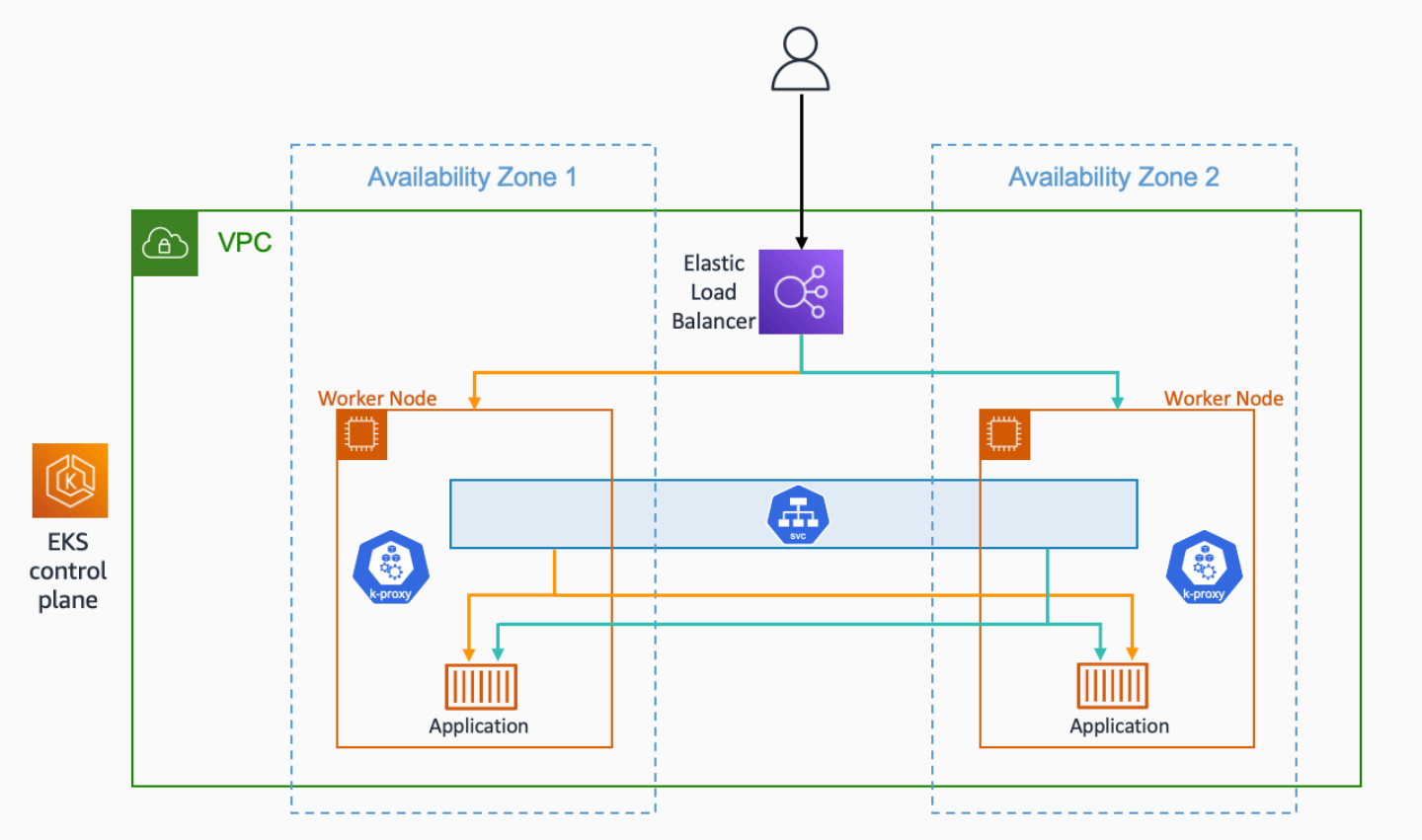
如果您使用 EKS 自動模式，則會自動為您提供 AWS Load Balancer 控制器；不需要安裝。

選擇Load Balancer目標類型

使用 IP Target-Type 將 Pod 註冊為目標

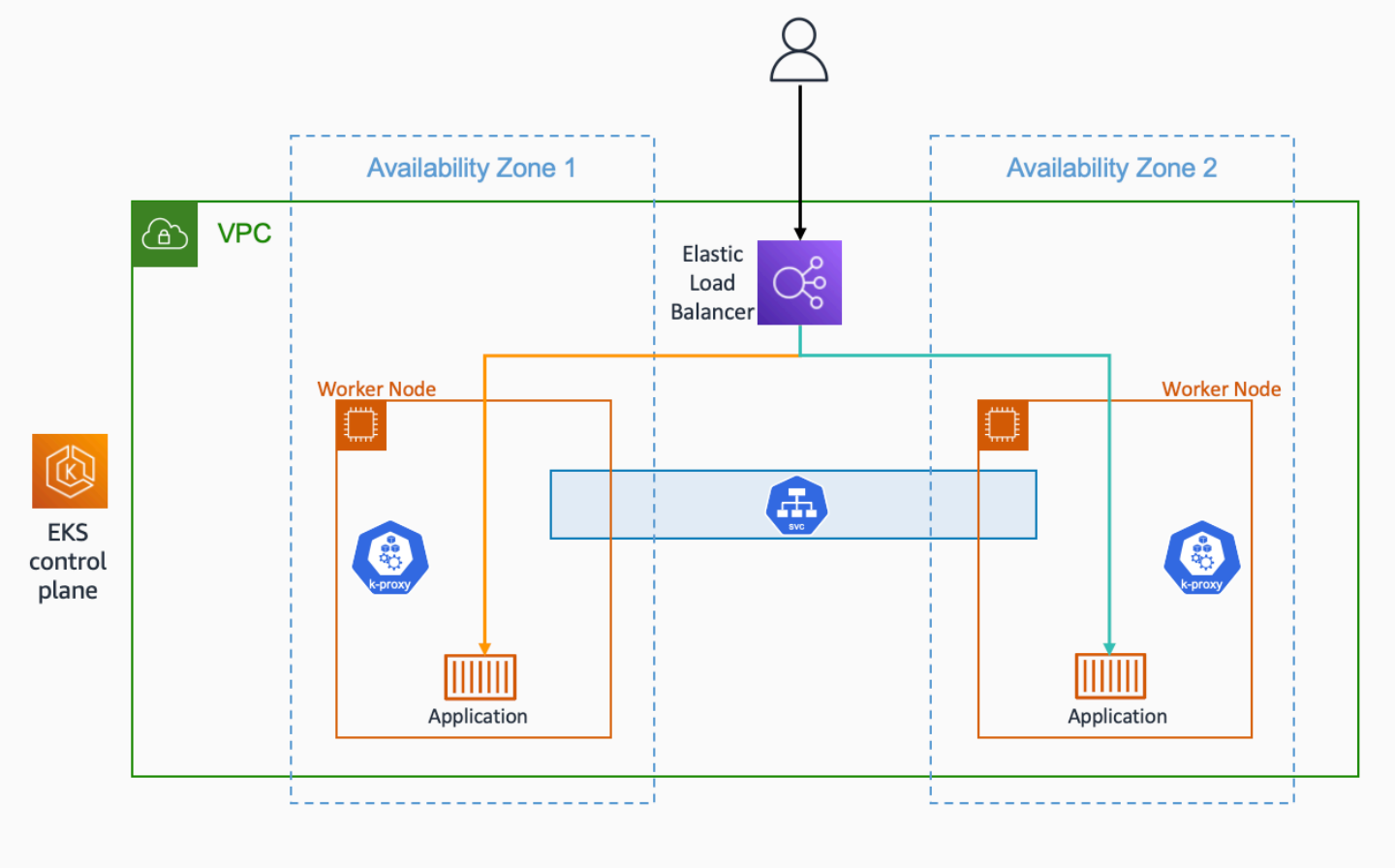
AWS Elastic Load Balancer：Network & Application，將接收的流量傳送至目標群組中的已註冊目標。對於 EKS 叢集，您可以在目標群組中註冊 2 種類型的目標：執行個體和 IP，使用哪種目標類型會影響註冊的內容，以及如何將流量從 Load Balancer 路由到 Pod。根據預設，AWS Load Balancer 控制器會使用「執行個體」類型註冊目標，而此目標將是工作者節點的 IP 和 NodePort，其含意包括：

- 來自Load Balancer的流量將轉送至 NodePort 上的工作者節點，這會由 iptables 規則處理（由節點上執行的 kube-proxy 設定），並轉送至 ClusterIP 上的服務（仍在節點上），最後服務會隨機選取已註冊的 Pod，並將流量轉送至該節點。此流程涉及多個躍點，而且可能會產生額外的延遲，特別是因為服務有時會選取在另一個工作者節點上執行的 Pod，而該節點也可能位於另一個可用區域。
- 由於Load Balancer會將工作者節點註冊為其目標，這表示傳送至目標的運作狀態檢查不會直接由 Pod 接收，但由其 NodePort 上的工作者節點接收，且運作狀態檢查流量將遵循上述相同的路徑。
- 監控和故障診斷更為複雜，因為Load Balancer轉送的流量不會直接傳送到 Pod，而且您必須謹慎地將工作者節點上收到的封包與服務 ClusterIP 建立關聯，最終讓 Pod 能夠完全 end-to-end 掌握封包路徑，以進行適當的故障診斷。



相反地，如果您將目標類型設定為「IP」，我們建議隱含內容如下：

- 來自Load Balancer的流量將直接轉送到 Pod，這簡化了網路路徑，因為它繞過先前的工作者節點和服務叢集 IP 額外跳轉，它減少了如果服務將流量轉送到另一個可用區域中的 Pod，最後移除工作者節點上的 iptables 規則額外負荷處理，否則會發生的延遲。
- Pod 會直接接收和回應 Load Balancer 的運作狀態檢查，這表示目標狀態「運作狀態良好」或「運作狀態不良」是 Pod 運作狀態的直接表示。
- 監控和故障診斷更容易，任何用於擷取封包 IP 地址的工具都會直接顯示Load Balancer與其來源和目的地欄位中 Pod 之間的雙向流量。



若要建立使用您新增之 IP 目標的 AWS Elastic Load Balancing：

- `alb.ingress.kubernetes.io/target-type: ip` 設定 Kubernetes 輸入時 (Application Load Balancer) 資訊清單的 註釋
- `service.beta.kubernetes.io/aws-load-balancer-nlb-target-type: ip` 設定類型 LoadBalancer (Network Load Balancer) 的 Kubernetes Service 時， 會註釋服務資訊清單。

設定Load Balancer運作狀態檢查

雖然 Kubernetes 提供自己的運作狀態檢查機制 （在下一節中詳細說明），但我們建議您實作 ELB 運作狀態檢查，做為在 Kubernetes 控制平面外部運作的補充保護措施。此獨立 layer 會在下列期間持續監控您的應用程式：

- Kubernetes 控制平面中斷
- 探查執行延遲
- kubelet 和 Pod 之間的網路分割區

對於在上述案例中需要最大可用性加速復原的關鍵工作負載，ELB 運作狀態檢查提供與 Kubernetes 原生機制一起運作的基本安全網，而不是替代。

若要在 ELB 上設定和微調運作狀態檢查，您必須使用 Kubernetes 服務或輸入資訊清單中的註釋，並由服務控制器或 AWS Load Balancer 控制器進行協調。

可用性和 Pod 生命週期

在應用程式升級期間，您必須確保您的應用程式隨時可用於處理請求，讓使用者不會遇到任何停機時間。在此案例中，一個常見的挑戰是在 Kubernetes 層和基礎設施之間同步工作負載的可用性狀態，例如外部負載平衡器。以下幾節重點介紹處理此類案例的最佳實務。

> **Note**
>
> 以下說明是以 EndpointSlices 為基礎，因為它是 Kubernetes 中端點的建議替換項目。在以下涵蓋的情境中，兩者之間的差異微乎其微。AWS Load Balancer 控制器預設會使用端點，您可以透過在控制器上啟用 enable-endpoint-sliceflag 來啟用 EndpointSlices。

使用運作狀態檢查

Kubernetes 預設會執行程序運作狀態檢查，其中節點上的 kubelet 程序會驗證容器的主要程序是否正在執行。如果沒有，則預設會重新啟動該容器。不過，您也可以設定 Kubernetes 探查來識別容器程序何時執行但處於死鎖狀態，或應用程式是否已成功啟動。探查可以基於 exec、grpc、httpGet 和 tcpSocket 機制。根據探查的類型和結果，可以重新啟動容器。

請參閱下方附錄一節中的 Pod 建立，以重新檢視 Pod 建立程序中的事件順序。

使用整備探查

根據預設，當 Pod 中的所有容器執行 Pod 條件時，會被視為「就緒」。不過，應用程式可能仍然無法處理用戶端請求。例如，應用程式可能需要從外部資源提取一些資料或組態，才能處理請求。在這種狀態下，您不會想要終止應用程式，也不會轉送任何請求。整備探查可讓您確保 Pod 不會被視為「就緒」，這表示在探查結果為 之前，不會將其新增至 EndpointSlice 物件`success`。另一方面，如果探測進一步失敗，則會從 EndpointSlice 物件中移除 Pod。您可以在每個容器的 Pod 資訊清單中設定整備探查。每個節點上的 `kubelet` 程序都會對該節點上的容器執行整備探查。

使用 Pod 整備閘道

整備探查的一個方面是，其中沒有外部意見回饋/影響機制，節點上的 kubelet 程序會執行探查並定義探查的狀態。這不會影響 Kubernetes 層中微服務本身之間的請求（東西流量），因為 EndpointSlice 控制器會將端點 (Pod) 清單保持在最新狀態。為什麼和何時需要外部機制，然後是 ？

當您使用 Load Balancer 或 Kubernetes Ingress（適用於北向 - 南向流量）的 Kubernetes Service 類型公開應用程式時，必須將個別 Kubernetes Service 的 Pod IPs 清單傳播至外部基礎設施負載平衡器，以便負載平衡器也具有最新的清單目標。AWS Load Balancer 控制器會橋接此處的間隙。當您使用 AWS Load Balancer 控制器並利用 `target group: IP` 時，就像 `kube-proxy` AWS Load Balancer 控制器也會收到更新（透過 `watch`），然後與 ELB API 通訊，以設定並開始將 Pod IP 註冊為 ELB 上的目標。

當您執行部署的滾動更新時，會建立新的 Pod，並在新 Pod 的條件為「就緒」時立即終止舊/現有的 Pod。在此過程中，Kubernetes EndpointSlice 物件的更新速度會比 ELB 將新 Pod 註冊為目標所需的時間更快，請參閱目標註冊。在短時間內，您可能會在 Kubernetes 層與可捨棄用戶端請求的基礎設施層之間發生狀態不符。在 Kubernetes layer 內的這段期間，新的 Pod 已準備好處理請求，但從 ELB 的角度來看，它們不是。

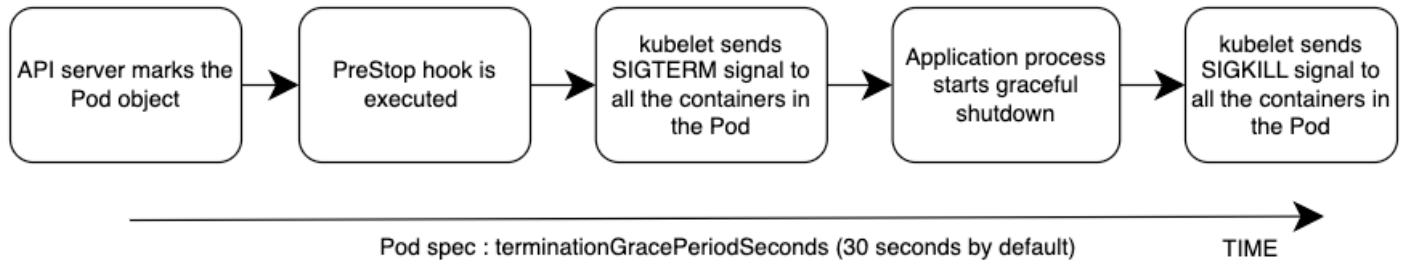
Pod 整備閘道可讓您定義 Pod 條件視為「就緒」之前必須符合的其他需求。在 AWS ELB 的情況下，AWS Load Balancer 控制器會監控 AWS ELB 上目標 (Pod) 的狀態，一旦目標註冊完成且其狀態變成「運作狀態」，控制器就會將 Pod 的條件更新為「就緒」。透過此方法，您可以根據外部網路的狀態影響 Pod 條件，這是 AWS ELB 上的目標狀態。Pod 整備閘道對於滾動更新案例至關重要，因為它可讓您防止部署的滾動更新終止舊的 Pod，直到新建立的 Pod 目標狀態在 AWS ELB 上變成「運作狀態」為止。

正常關閉應用程式

您的應用程式應該透過啟動正常關機來回應 SIGTERM 訊號，以便用戶端不會經歷任何停機時間。這表示您的應用程式應執行清除程序，例如儲存資料、關閉檔案描述項、關閉資料庫連線、正常完成傳輸中請求，以及及時結束以滿足 Pod 終止請求。您應該將寬限期設定為足夠長的時間，以便清除可以完成。若要了解如何回應 SIGTERM 訊號，您可以參考您用於應用程式之個別程式設計語言的資源。

如果您的應用程式在收到 SIGTERM 訊號時無法正常關閉，或忽略/未接收訊號，您可以改為利用 PreStop 掛鉤啟動正常的應用程式關閉。在傳送 SIGTERM 訊號之前立即執行預載勾點，它可以執行任意操作，而無需在應用程式程式碼本身中實作這些操作。

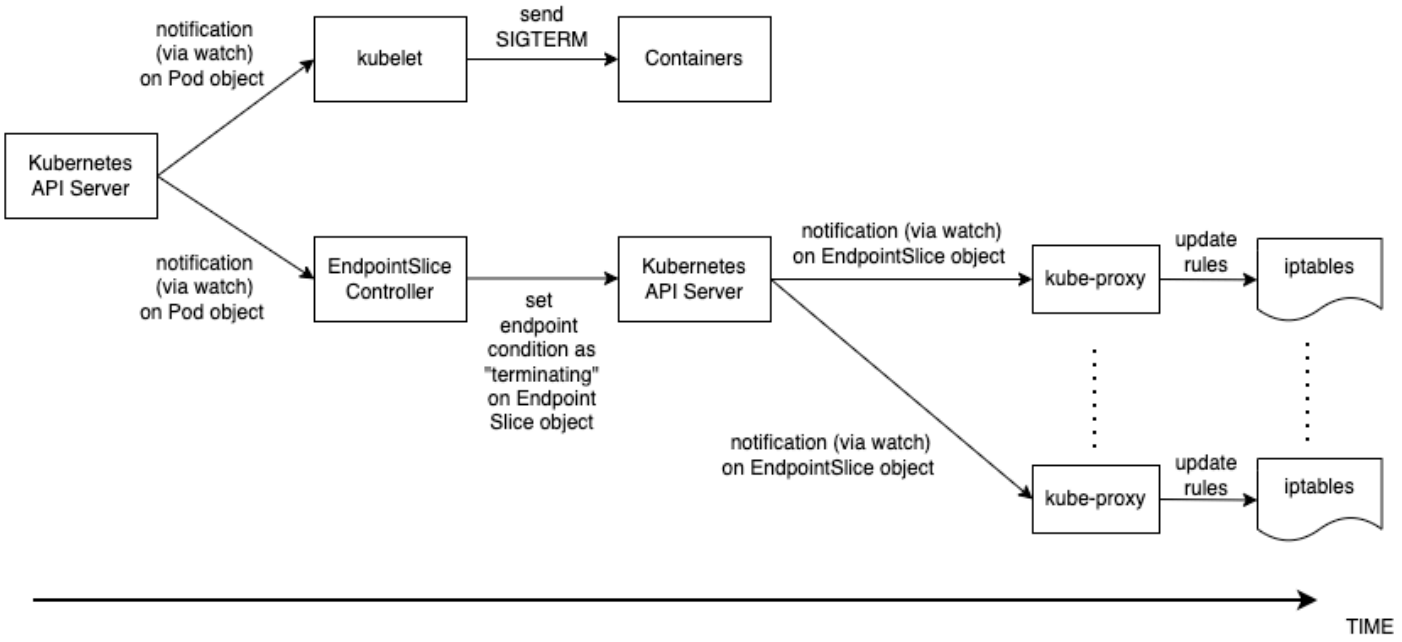
整體事件順序如下圖所示。注意：無論應用程式正常關閉程序的結果或 PreStop 掛鉤的結果為何，應用程式容器最終都會在寬限期結束時透過 SIGKILL 終止。



請參閱下方附錄一節中的 Pod 刪除，以重新檢視 Pod 刪除程序中的事件順序。

優雅地處理用戶端請求

Pod 刪除中的事件順序與建立 Pod 不同。建立 Pod 時，會在 Kubernetes API 中kubelet更新 Pod IP，而且只會更新 EndpointSlice 物件。另一方面，當 Pod 正在終止時，Kubernetes API 會同時通知 kubelet 和 EndpointSlice 控制器。仔細檢查下圖，其中顯示事件的順序。



狀態從 API 伺服器傳播到上述節點上 iptables 規則的方式，會建立有趣的競爭條件。由於容器接收 SIGKILL 訊號的機率很高，比每個節點上的 kube-proxy 更早，因此會更新本機 iptables 規則。在這種情況下，值得提及的兩個案例是 ：

- 如果您的應用程式在收到 SIGTERM 時立即且暗中捨棄處理中的請求和連線，這表示用戶端會看到整個位置的 50 倍錯誤。

- 即使您的應用程式確保在收到 SIGTERM 時，所有處理中的請求和連線都會完全處理，在寬限期內，新的用戶端請求仍會傳送到應用程式容器，因為 iptables 規則可能尚未更新。在清除程序關閉容器上的伺服器通訊端之前，這些新請求將產生新的連線。當寬限期結束時，這些連線會在 SIGTERM 之後建立，此時會無條件捨棄，因為 SIGKILL 已傳送。

在 Pod 規格中設定足夠長的寬限期可以解決此挑戰，但取決於傳播延遲和實際用戶端請求的數量，很難預測應用程式正常關閉連線所需的時間。因此，此處不完美但最可行的方法是使用 PreStop 勾點來延遲 SIGTERM 訊號，直到更新 iptables 規則，以確保不會將新的用戶端請求傳送至應用程式，而是只有現有的連線會繼續。PreStop hook 可以是簡單的 Exec 處理常式，例如 `sleep 10`。

當您使用 AWS Load Balancer 控制器使用 Load Balancer 的 Kubernetes Service 類型或 Kubernetes Ingress（適用於北向 - 南向流量）公開應用程式並利用 `target group: IP` 時，上述行為和建議同樣適用。 Load Balancer 因為就像 `kube-proxy` AWS Load Balancer 控制器一樣，它也會收到 EndpointSlice 物件的更新（透過監看），然後與 ELB API 通訊，以開始從 ELB 取消註冊 Pod IP。不過，根據 Kubernetes API 或 ELB API 的負載，這也可能需要一些時間，而且 SIGTERM 可能已在很久之前傳送到應用程式。一旦 ELB 開始取消註冊目標，它會停止傳送請求到該目標，因此應用程式不會收到任何新的請求，ELB 也會啟動取消註冊延遲，預設為 300 秒。在取消註冊過程中，目標`draining`基本上是 ELB 等待處理中的請求/與該目標的現有連線耗盡。一旦取消註冊延遲過期，目標就會未使用，且對該目標的任何傳輸中請求都會被強制捨棄。

使用 Pod 中斷預算

為您的應用程式設定 Pod 中斷預算 (PDB)。PDBlimits複寫應用程式因自願中斷而同時停機的 Pod 數量。它可確保在 StatefulSet 或部署中保持最低數量或百分比的 Pod 可用。例如，以規定人數為基礎的應用程式需要確保執行的複本數量永遠不會低於規定人數所需的數量。或者，Web 前端可以確保提供負載的複本數量永遠不會低於總數的特定百分比。PDB 將保護應用程式免於執行如節點耗盡或新部署版本推出等動作。請記住，PDB 不會保護應用程式免受非自願中斷，例如節點作業系統故障或網路連線中斷。如需詳細資訊，請參閱 Kubernetes 文件中的為您的應用程式指定中斷預算。

參考

- KubeCon 歐洲 2019 工作階段 - 準備好了嗎？ 深入了解適用於服務運作狀態的 Pod 準備階段
- 書籍 - Kubernetes 實際運作
- AWS 部落格 - 如何在 EKS 上使用 ALB 快速擴展應用程式（而不會遺失流量）

附錄

Pod 建立

在部署 Pod 的案例中，必須了解事件序列是什麼，然後它變得正常/準備好接收和處理用戶端請求。讓我們討論事件的順序。

1. Pod 是在 Kubernetes 控制平面上建立的（即 透過 kubectl 命令、部署更新或擴展動作）。
2. `kube-scheduler` 會將 Pod 指派給叢集中的節點。
3. 在指派節點上執行的 kubelet 程序會收到更新（透過 `watch`），並與容器執行時間通訊，以啟動 Pod 規格中定義的容器。
4. 當容器開始執行時， kubelet 會將 Pod 條件更新為 Kubernetes API `Ready`中的 Pod 物件。
5. EndpointSlice 控制器會收到 Pod 條件更新（透過 `watch`），並將 Pod IP/Port 做為新端點新增至個別 Kubernetes 服務的 EndpointSlice 物件 (Pod IPs 清單）。
6. 每個節點上的 kube-proxy 程序都會收到 EndpointSlice 物件的更新（透過 `watch`），然後使用新的 Pod IP/連接埠更新每個節點上的 iptables 規則。

Pod 刪除

如同建立 Pod 一樣，您必須了解 Pod 刪除期間的事件順序。讓我們討論事件的序列。

1. Pod 刪除請求會傳送至 Kubernetes API 伺服器（即 透過`kubectl`命令、部署更新或擴展動作）。
2. Kubernetes API 伺服器會在 Pod 物件中設定 deleteTimestamp 欄位，以啟動寬限期，預設為 30 秒。 deletionTimestamp （寬限期可透過 在 Pod 規格中設定`terminationGracePeriodSeconds`)
3. 在節點上執行`kubelet`的程序會接收 Pod 物件的更新（透過監看），並傳送 SIGTERM 訊號以處理該 Pod 中每個容器內的識別符 1 (PID 1)。然後，它會監看 `terminationGracePeriodSeconds`。
4. EndpointSlice 控制器也會從步驟 2 接收更新（透過 `watch`），並將端點條件設定為個別 Kubernetes 服務的 EndpointSlice 物件 (Pod IPs 清單） 中的「終止」。
5. 每個節點上的 kube-proxy 程序會在 EndpointSlice 物件上接收更新（透過 `watch`），然後 kube-proxy 會更新每個節點上的 iptables 規則，以停止將用戶端請求轉送至 Pod。
6. 當 `terminationGracePeriodSeconds`過期時， `kubelet`會將 SIGKILL 訊號傳送至 Pod 中每個容器的父程序，並強制終止它們。

7. TheEndpointSlice控制器會從 EndpointSlice 物件移除端點。
8. API 伺服器會刪除 Pod 物件。

監控 EKS 工作負載的網路效能問題

監控 DNS 限流問題的 CoreDNS 流量

執行 DNS 密集型工作負載有時可能會因為 DNS 限流而發生間歇性 CoreDNS 失敗，這可能會影響偶爾發生 UnknownHostException 錯誤的應用程式。

Deployment for CoreDNS 具有反親和性政策，指示 Kubernetes 排程器在叢集中的不同工作者節點上執行 CoreDNS 執行個體，也就是說，它應該避免在相同的工作者節點上共置複本。這可有效減少每個網路界面的 DNS 查詢數量，因為來自每個複本的流量會透過不同的 ENI 路由。如果您注意到 DNS 查詢因為每秒 1024 個封包的限制而受到調節，您可以 1) 嘗試增加 CoreDNS 複本的數量，或 2) 實作 NodeLocal DNSCache。如需詳細資訊，請參閱監控 CoreDNS 指標。

挑戰

- 封包捨棄會在幾秒鐘內發生，我們很難正確監控這些模式，以判斷 DNS 調節是否實際發生。
- DNS 查詢會在彈性網路介面層級調節。因此，限流查詢不會出現在查詢日誌中。
- 流量日誌不會擷取所有 IP 流量。例如 由執行個體在與 Amazon DNS 伺服器聯絡時產生的流量。如果您使用自己的 DNS 伺服器，則會記錄該 DNS 伺服器的所有流量

解決方案

識別工作者節點中 DNS 限流問題的簡單方法是擷取 `linklocal_allowance_exceeded` 指標。linklocal_allowance_exceeded 是由於流量到本機代理服務的 PPS 超過網路介面的最大值而捨棄的封包數。這會影響 DNS 服務、執行個體中繼資料服務和 Amazon Time Sync Service 的流量。除了即時追蹤此事件，我們也可以將此指標串流到 Amazon Managed Service for Prometheus，並在 Amazon Managed Grafana 中將其視覺化

使用 Conntrack 指標監控 DNS 查詢延遲

另一個有助於監控 CoreDNS 限流/查詢延遲的指標是 `conntrack_allowance_available`和 `conntrack_allowance_exceeded`。超過連線追蹤額度所造成的連線失敗，其影響可能比超過其他額度所造成的影響更大。當依賴 TCP 傳輸資料時，由於超過 EC2 執行個體網路額度而排入佇列或捨

棄的封包，例如頻寬、PPS 等，通常會因為 TCP 的擁塞控制功能而正常處理。受影響的流程將減慢速度，而遺失的封包將重新傳輸。不過，當執行個體超過其連線追蹤額度時，在關閉一些現有連線以騰出空間進行新連線之前，無法建立新的連線。

conntrack_allowance_available 和 conntrack_allowance_exceeded協助客戶監控連線追蹤額度，因每個執行個體而異。這些網路效能指標可讓客戶了解執行個體聯網額度時佇列或捨棄的封包數量，例如網路頻寬、Packets-Per-Second數 (PPS)、追蹤的連線和連結本機服務存取 (Amazon DNS、執行個體中繼資料服務、Amazon Time Sync)

conntrack_allowance_available 是執行個體在達到該執行個體類型的連線追蹤額度之前可建立的追蹤連線數目（僅支援 nitro 型執行個體）。 conntrack_allowance_exceeded是由於連線追蹤超過執行個體的最大值而無法建立新的連線而捨棄的封包數目。

其他重要的網路效能指標

其他重要的網路效能指標包括：

bw_in_allowance_exceeded（指標的理想值應為零）是佇列和/或捨棄的封包數量，因為傳入彙總頻寬超過執行個體的上限

bw_out_allowance_exceeded（指標的理想值應為零）是佇列和/或捨棄的封包數量，因為傳出彙總頻寬超過執行個體的上限

pps_allowance_exceeded（指標的理想值應為零）是由於雙向 PPS 超過執行個體的最大值而佇列和/或捨棄的封包數量

擷取指標以監控工作負載的網路效能問題

Elastic Network Adapter (ENA) 驅動程式會從啟用它們的執行個體發佈上述的網路效能指標。所有網路效能指標都可以使用 CloudWatch 代理程式發佈至 CloudWatch。如需詳細資訊，請參閱部落格。

現在讓我們擷取上述指標，將其存放在 Amazon Managed Service for Prometheus 中，並使用 Amazon Managed Grafana 視覺化

先決條件

- ethtool - 確保工作者節點已安裝 ethtool
- 在您的 AWS 帳戶中設定的 AMP 工作區。如需說明，請參閱《AMP 使用者指南》中的建立工作區。
- Amazon Managed Grafana 工作區

部署 Prometheus Node Exporter

為了收集工作者節點的 ethtool 指標，我們需要部署收集器，以收集指標並以 Prometheus 格式公開它們。下列命令將新增 prometheus-community Helm 儲存庫，並在啟用ethtool收集器的情況下安裝prometheus-node-exporter圖表。

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm upgrade -i prometheus-node-exporter prometheus-community/prometheus-node-exporter
 \
  --set extraArgs[0]="--collector.ethtool" \
  --set extraArgs[1]="--collector.ethtool.device-include=(eth|em|eno|ens|enp)[0-9s]+" \
  --set extraArgs[2]="--collector.ethtool.metrics-include=.*" \
  --set podAnnotations."prometheus\.io/scrape='true'" \
  --set podAnnotations."prometheus\.io/port='9100'"
```

部署 ADOT 收集器以抓取 ethtool 指標，並存放在 Amazon Managed Service for Prometheus 工作區

安裝 AWS Distro for OpenTelemetry (ADOT) 的每個叢集都必須具有此角色，才能授予 AWS 服務帳戶將指標存放至 Amazon Managed Service for Prometheus 的許可。按照以下步驟，使用 IRSA 建立您的 IAM 角色，並將其關聯至您的 Amazon EKS 服務帳戶：

```
eksctl create iamserviceaccount --name adot-collector --namespace default
 --cluster <CLUSTER_NAME> --attach-policy-arn arn:aws:iam::aws:policy/
AmazonPrometheusRemoteWriteAccess --attach-policy-arn arn:aws:iam::aws:policy/
AWSXrayWriteOnlyAccess --attach-policy-arn arn:aws:iam::aws:policy/
CloudWatchAgentServerPolicy --region <REGION> --approve  --override-existing-
serviceaccounts
```

讓我們部署 ADOT 收集器，從 prometheus ethtool 匯出工具中抓取 metrcis，並將其存放在 Amazon Managed Service for Prometheus 中

下列程序使用範例 YAML 檔案，並將部署做為模式值。這是預設模式，且部署 ADOT 收集器的程序與獨立應用程式類似。此組態會從範例應用程式和從叢集上的 Pod 抓取的 Amazon Managed Service for Prometheus 指標接收 OTLP 指標

```
curl -o collector-config-amp.yaml https://raw.githubusercontent.com/aws-observability/
aws-otel-community/master/sample-configs/operator/collector-config-amp.yaml
```

在 collector-config-amp.yaml 中，將下列項目取代為您自己的值：

- 模式：部署
- serviceAccount：adot-collector
- 端點：<YOUR_REMOTE_WRITE_ENDPOINT>
- 區域：<YOUR_AWS_REGION>
- 名稱：adot-collector

```
kubectl apply -f collector-config-amp.yaml
```

部署 adot 收集器後，指標將成功儲存在 Amazon Prometheus 中

在 Amazon Managed Service for Prometheus 中設定警示管理員以傳送通知

您可以在 Amazon Managed Service for Prometheus 中使用警示管理員來設定重要警示的警示規則，然後您可以將通知傳送至 Amazon SNS 主題。讓我們設定記錄規則和提醒規則，以檢查到目前為止討論的指標。

我們將使用 Amazon Managed Service for Prometheus 的 ACK 控制器來佈建提醒和記錄規則。

讓我們部署 Amazon Managed Service for Prometheus 服務的 ACL 控制器：

```
export SERVICE=prometheusservice
export RELEASE_VERSION=`curl -sL https://api.github.com/repos/aws-controllers-k8s/
$SERVICE-controller/releases/latest | grep '"tag_name":' | cut -d'"' -f4`
export ACK_SYSTEM_NAMESPACE=ack-system
export AWS_REGION=us-east-1
aws ecr-public get-login-password --region us-east-1 | helm registry login --username
 AWS --password-stdin public.ecr.aws
helm install --create-namespace -n $ACK_SYSTEM_NAMESPACE ack-$SERVICE-controller \
oci://public.ecr.aws/aws-controllers-k8s/$SERVICE-chart --version=$RELEASE_VERSION --
set=aws.region=$AWS_REGION
```

執行 命令，稍後您應該會看到下列訊息：

```
You are now able to create Amazon Managed Service for Prometheus (AMP) resources!

The controller is running in "cluster" mode.

The controller is configured to manage AWS resources in region: "us-east-1"
```

```
The ACK controller has been successfully installed and ACK can now be used to provision
 an Amazon Managed Service for Prometheus workspace.
```

現在讓我們建立 yaml 檔案來佈建警示管理員定義和規則群組。將下列檔案儲存為 `rulegroup.yaml`

```
apiVersion: prometheusservice.services.k8s.aws/v1alpha1
kind: RuleGroupsNamespace
metadata:
   name: default-rule
spec:
   workspaceID: <Your WORKSPACE-ID>
   name: default-rule
   configuration: |
     groups:
     - name: ppsallowance
       rules:
       - record: metric:pps_allowance_exceeded
         expr: rate(node_net_ethtool{device="eth0",type="pps_allowance_exceeded"}[30s])
       - alert: PPSAllowanceExceeded
         expr: rate(node_net_ethtool{device="eth0",type="pps_allowance_exceeded"}
 [30s]) > 0
         labels:
           severity: critical

         annotations:
           summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
           description: "PPSAllowanceExceeded is greater than 0"
     - name: bw_in
       rules:
       - record: metric:bw_in_allowance_exceeded
         expr: rate(node_net_ethtool{device="eth0",type="bw_in_allowance_exceeded"}
[30s])
       - alert: BWINAllowanceExceeded
         expr: rate(node_net_ethtool{device="eth0",type="bw_in_allowance_exceeded"}
 [30s]) > 0
         labels:
           severity: critical

         annotations:
           summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
           description: "BWInAllowanceExceeded is greater than 0"
```

```
    - name: bw_out
      rules:
      - record: metric:bw_out_allowance_exceeded
        expr: rate(node_net_ethtool{device="eth0",type="bw_out_allowance_exceeded"}
[30s])
      - alert: BWOutAllowanceExceeded
        expr: rate(node_net_ethtool{device="eth0",type="bw_out_allowance_exceeded"}
 [30s]) > 0
        labels:
          severity: critical

        annotations:
          summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
          description: "BWoutAllowanceExceeded is greater than 0"
    - name: conntrack
      rules:
      - record: metric:conntrack_allowance_exceeded
        expr: rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}
[30s])
      - alert: ConntrackAllowanceExceeded
        expr: rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}
 [30s]) > 0
        labels:
          severity: critical

        annotations:
          summary: Connections dropped due to total allowance exceeding for the
 (instance {{ $labels.instance }})
          description: "ConnTrackAllowanceExceeded is greater than 0"
    - name: linklocal
      rules:
      - record: metric:linklocal_allowance_exceeded
        expr: rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}
[30s])
      - alert: LinkLocalAllowanceExceeded
        expr: rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}
 [30s]) > 0
        labels:
          severity: critical

        annotations:
          summary: Packets dropped due to PPS rate allowance exceeded for local
 services  (instance {{ $labels.instance }})
```

```
            description: "LinkLocalAllowanceExceeded is greater than 0"
```

將 WORKSPACE-ID 取代為您正在使用之工作區的工作區 ID。

現在讓我們設定警示管理員定義。將以下 fie 儲存為 `alertmanager.yaml`

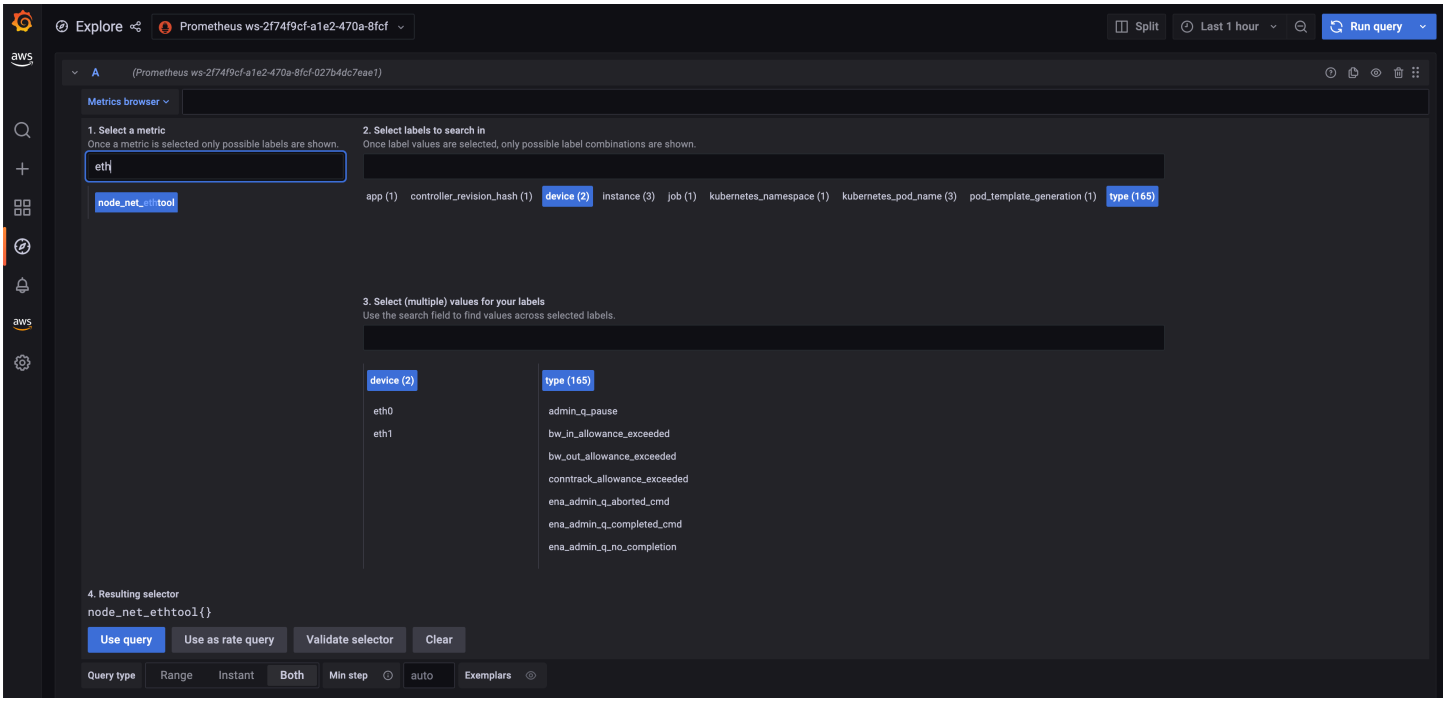
```
apiVersion: prometheusservice.services.k8s.aws/v1alpha1
kind: AlertManagerDefinition
metadata:
  name: alert-manager
spec:
  workspaceID: <Your WORKSPACE-ID >
  configuration: |
    alertmanager_config: |
      route:
         receiver: default_receiver
       receivers:
       - name: default_receiver
          sns_configs:
          - topic_arn: TOPIC-ARN
            sigv4:
              region: REGION
            message: |
              alert_type: {{ .CommonLabels.alertname }}
              event_type: {{ .CommonLabels.event_type }}
```

將 You WORKSPACE-ID 取代為新工作區的工作區 ID，將 TOPIC-ARN 取代為您要傳送提醒之 Amazon Simple Notification Service 主題的 ARN，並將 REGION 取代為工作負載的目前區域。請確定您的工作區具有傳送訊息至 Amazon SNS 的許可。

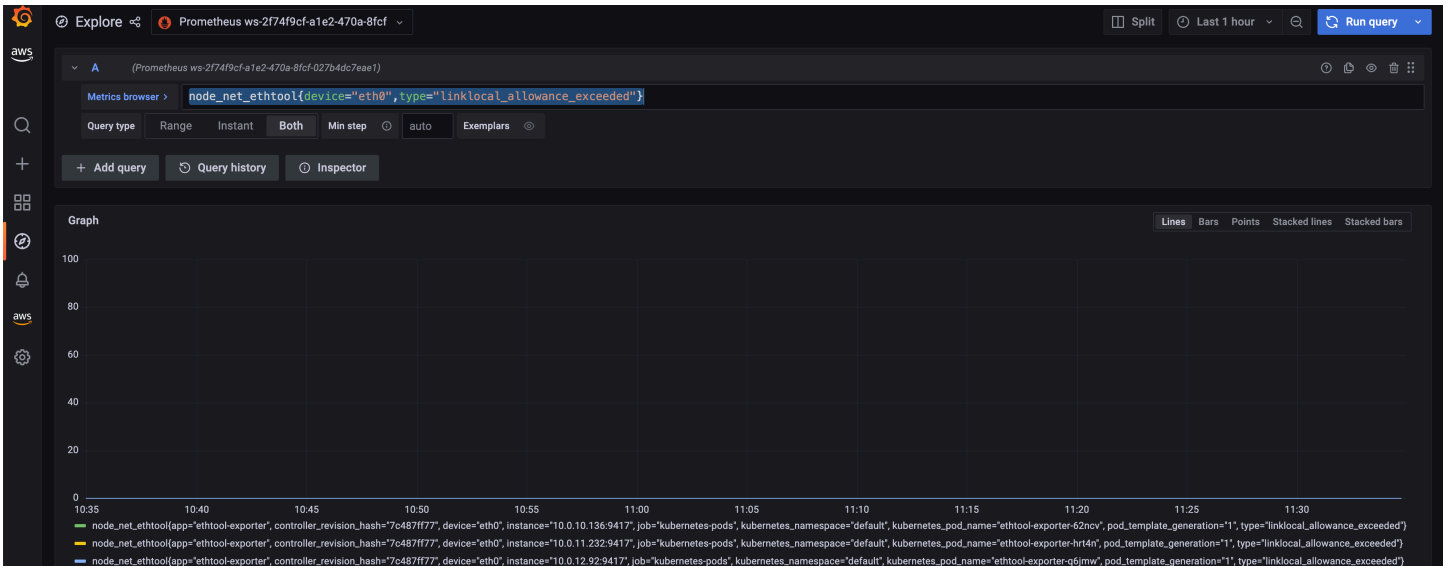
在 Amazon Managed Grafana 中視覺化 ethtool 指標

讓我們視覺化 Amazon Managed Grafana 中的指標並建置儀表板。在 Amazon Managed Grafana 主控台中將 Amazon Managed Service for Prometheus 設定為資料來源。如需說明，請參閱新增 Amazon Prometheus 做為資料來源

現在讓我們來探索 Amazon Managed Grafana 中的指標：按一下探索按鈕，並搜尋 ethtool：



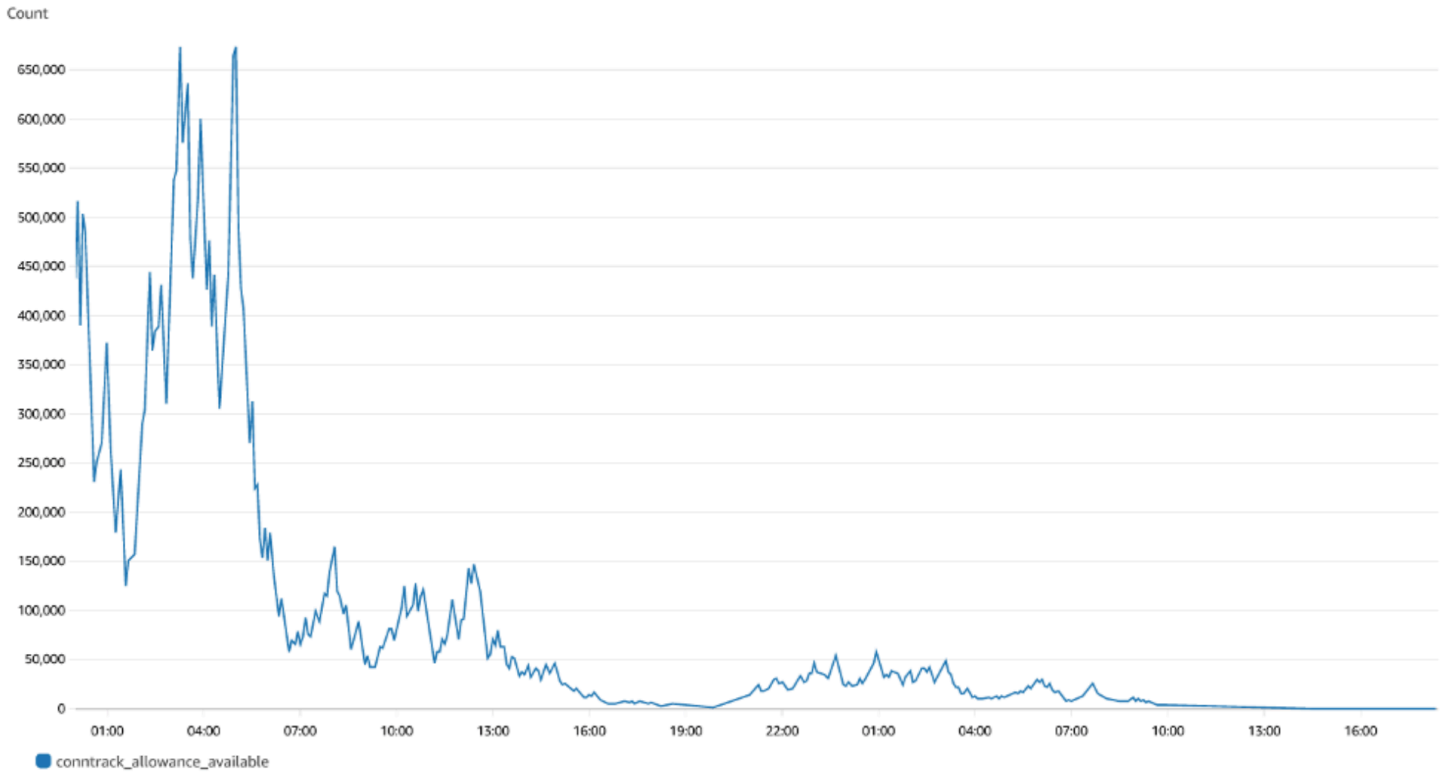
讓我們使用查詢 建置 linklocal_allowance_exceededed 指標的儀表板`rate(node_net_ethtool{device="eth0",type="linklocal_allowance_exceeded"}[30s])`。這將導致以下儀表板。



我們可以看到沒有封包遭到捨棄，因為值為零。

讓我們使用查詢 建置 conntrack_allowance_exceededed 指標的儀表板`rate(node_net_ethtool{device="eth0",type="conntrack_allowance_exceeded"}[30s])`。這將導致以下儀表板。

指標conntrack_allowance_exceeded可在 CloudWatch 中視覺化，前提是您執行此處所述的 cloudwatch 代理程式。CloudWatch 中產生的儀表板如下所示：



我們可以看到沒有封包遭到捨棄，因為值為零。如果您使用 Nitro 型執行個體，您可以為 建立類似的儀表板，conntrack_allowance_available並主動監控 EC2 執行個體中的連線。您可以在 Amazon Managed Grafana 中設定提醒，將通知傳送至 Slack、SNS、Pagerduty 等，以進一步擴展此範圍。

在 IPVS 模式下執行 kube-proxy

IP Virtual Server (IPVS) 模式下的 EKS 可解決在舊版 iptables 模式下執行超過 1，000 個 kube-proxy 服務的大型叢集時經常出現的網路延遲問題。此效能問題是針對每個封包依序處理 iptables 封包篩選規則的結果。此延遲問題已在 nftables 中解決，這是 iptable 的後續版本。不過，截至撰寫本文時，kube-proxy 仍在開發中，以利用 nftable。若要解決此問題，您可以將叢集設定為`kube-proxy`在 IPVS 模式下執行。

概觀

IPVS 自 Kubernetes 1.11 版以來即為 GA，使用雜湊表而非線性搜尋來處理封包，為具有數千個節點和服務的叢集提供效率。IPVS 專為負載平衡而設計，使其成為 Kubernetes 網路效能問題的適當解決方案。

IPVS 提供多種選項，可將流量分配到後端 Pod。您可以在官方 Kubernetes 文件中找到每個選項的詳細資訊，但簡單的清單如下所示。Round Robin 和 Least Connections 是 Kubernetes 中 IPVS 負載平衡選項最常見的選項之一。

```
- rr (Round Robin)
- wrr (Weighted Round Robin)
- lc (Least Connections)
- wlc (Weighted Least Connections)
- lblc (Locality Based Least Connections)
- lblcr (Locality Based Least Connections with Replication)
- sh (Source Hashing)
- dh (Destination Hashing)
- sed (Shortest Expected Delay)
- nq (Never Queue)
```

實作

在 EKS 叢集中啟用 IPVS 只需要幾個步驟。您需要做的第一件事是確保您的 EKS 工作者節點映像已安裝 Linux Virtual Server 管理`ipvsadm`套件。若要在 Amazon Linux 2023 等以 Fedora 為基礎的映像上安裝此套件，您可以在工作者節點執行個體上執行下列命令。

```
sudo dnf install -y ipvsadm
```

在以 Debian 為基礎的映像上，例如 Ubuntu，安裝命令看起來會像這樣。

```
sudo apt-get install ipvsadm
```

接著，您需要載入上列 IPVS 組態選項的核心模組。我們建議將這些模組寫入 /etc/modules-load.d/目錄內的檔案，使其在重新開機後仍然存活。

```
sudo sh -c 'cat << EOF > /etc/modules-load.d/ipvs.conf
ip_vs
ip_vs_rr
ip_vs_wrr
ip_vs_lc
ip_vs_wlc
ip_vs_lblc
ip_vs_lblcr
ip_vs_sh
ip_vs_dh
ip_vs_sed
ip_vs_nq
nf_conntrack
EOF'
```

您可以執行下列命令，在已執行 的機器上載入這些模組。

```
sudo modprobe ip_vs
sudo modprobe ip_vs_rr
sudo modprobe ip_vs_wrr
sudo modprobe ip_vs_lc
sudo modprobe ip_vs_wlc
sudo modprobe ip_vs_lblc
sudo modprobe ip_vs_lblcr
sudo modprobe ip_vs_sh
sudo modprobe ip_vs_dh
sudo modprobe ip_vs_sed
sudo modprobe ip_vs_nq
sudo modprobe nf_conntrack
```

> **Note**
>
> 強烈建議您透過使用者資料指令碼或任何執行的建置指令碼執行這些工作者節點步驟，做為工作者節點引導程序的一部分，以建置自訂工作者節點 AMI。

接著，您將設定叢集的 kube-proxy DaemonSet 以 IPVS 模式執行。方法是將 kube-proxy mode 設定為 ipvs，並將 ipvs scheduler設定為上述其中一個負載平衡選項，例如：rr適用於 Round Robin。

> **Warning**
>
> 這是破壞性變更，應該在非上班時間執行。我們建議在初始 EKS 叢集建立期間進行這些變更，以將影響降至最低。

您可以更新 EKS 附加元件，發出 AWS CLI kube-proxy 命令來啟用 IPVS。

```
aws eks update-addon --cluster-name $CLUSTER_NAME --addon-name kube-proxy \
  --configuration-values '{"ipvs": {"scheduler": "rr"}, "mode": "ipvs"}' \
  --resolve-conflicts OVERWRITE
```

或者，您也可以修改叢集中的 kube-proxy-config ConfigMap 來執行此操作。

```
kubectl -n kube-system edit cm kube-proxy-config
```

尋找下方的scheduler設定ipvs，並將 值設定為上述其中一個 ipvs 負載平衡選項，例如：rr適用於 Round Robin。尋找預設為 mode的設定iptables，並將值變更為 ipvs。任一選項的結果看起來都應該類似於以下組態。

```
  iptables:
    masqueradeAll: false
    masqueradeBit: 14
    minSyncPeriod: 0s
    syncPeriod: 30s
  ipvs:
    excludeCIDRs: null
    minSyncPeriod: 0s
    scheduler: "rr"
    syncPeriod: 30s
  kind: KubeProxyConfiguration
  metricsBindAddress: 0.0.0.0:10249
  mode: "ipvs"
  nodePortAddresses: null
  oomScoreAdj: -998
  portRange: ""
```

```
  udpIdleTimeout: 250ms
```

如果您的工作者節點在進行這些變更之前已加入叢集，您將需要重新啟動 kube-proxy DaemonSet。

```
kubectl -n kube-system rollout restart ds kube-proxy
```

驗證

您可以在其中一個工作者節點上發出下列命令，以驗證叢集和工作者節點是否在 IPVS 模式下執行。

```
sudo ipvsadm -L
```

您至少應該會看到類似以下的結果，顯示位於 的 Kubernetes API Server 服務項目，`10.100.0.1`以及位於 的 CoreDNS 服務項目`10.100.0.10`。

```
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  ip-10-100-0-1.us-east-1. rr
  -> ip-192-168-113-81.us-eas Masq        1      0          0
  -> ip-192-168-162-166.us-ea Masq        1      1          0
TCP  ip-10-100-0-10.us-east-1 rr
  -> ip-192-168-104-215.us-ea Masq        1      0          0
  -> ip-192-168-123-227.us-ea Masq        1      0          0
UDP  ip-10-100-0-10.us-east-1 rr
  -> ip-192-168-104-215.us-ea Masq        1      0          0
  -> ip-192-168-123-227.us-ea Masq        1      0          0
```

> **Note**
>
> 此範例輸出來自服務 IP 地址範圍為 的 EKS 叢集`10.100.0.0/16`。

EKS 可擴展性最佳實務

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

本指南提供擴展 EKS 叢集的建議。擴展 EKS 叢集的目標是最大化單一叢集可執行的工作量。與使用多個叢集相比，使用單一大型 EKS 叢集可以減少操作負載，但對多區域部署、租用戶隔離和叢集升級等項目具有取捨。在本文件中，我們將著重於如何透過單一叢集達到最大的可擴展性。

本指南的使用方式

本指南適用於負責在 AWS 中建立和管理 EKS 叢集的開發人員和管理員。它專注於一些一般的 Kubernetes 擴展實務，但它沒有自我管理的 Kubernetes 叢集，或在具有 EKS Anywhere 的 AWS 區域之外執行的叢集的詳細資訊。

每個主題都有簡短的概觀，後面接著大規模操作 EKS 叢集的建議和最佳實務。主題不需要以特定順序讀取，如果沒有測試和驗證它們在叢集中運作，則不應套用建議。

了解擴展維度

可擴展性與效能和可靠性不同，在規劃叢集和工作負載需求時應考慮這三項。隨著叢集擴展，需要監控它們，但本指南不會涵蓋監控最佳實務。EKS 可以擴展到大型，但您需要規劃如何將叢集擴展到超過 300 個節點或 5，000 個 Pod。這些不是絕對數字，但它們來自與多個使用者、工程師和支援專業人員合作本指南。

在 Kubernetes 中擴展是多維的，而且沒有任何特定設定或建議適用於各種情況。我們可以提供擴展指引的主要領域包括：

EKS 叢集中的 Kubernetes 控制平面包含 AWS 自動為您執行和擴展的所有服務（例如 Kubernetes API 伺服器）。擴展控制平面是 AWS 的責任，但負責任地使用控制平面是您的責任。

Kubernetes Data Plane 擴展處理叢集和工作負載所需的 AWS 資源，但它們位於 EKS 控制平面之外。包括 EC2 執行個體、kubelet 和儲存體在內的資源都需要隨著叢集擴展進行擴展。

叢集服務是在叢集內執行的 Kubernetes 控制器和應用程式，為您的叢集和工作負載提供功能。這些可以是 EKS 附加元件，以及您為合規和整合而安裝的其他服務或 Helm Chart。這些服務通常取決於工作負載，並且隨著您的工作負載擴展，您的叢集服務將需要隨著它們擴展。

工作負載是您擁有叢集的原因，應隨著叢集水平擴展。Kubernetes 中有工作負載的整合和設定，可協助叢集擴展。Kubernetes 抽象概念也有架構考量，例如命名空間和服務。

超大型擴展

如果您要將單一叢集擴展到超過 1，000 個節點或 50，000 個 Pod，我們希望與您交談。我們建議您聯絡您的支援團隊或技術客戶經理，與可協助您規劃和擴展超過本指南所提供資訊的專家聯絡。如果您選擇加入，Amazon EKS 可以在單一叢集中支援最多 100，000 個節點。

Kubernetes 控制平面

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

Kubernetes 控制平面包含 Kubernetes API Server、Kubernetes Controller Manager、Scheduler 和其他 Kubernetes 運作所需的元件。這些元件的可擴展性限制會根據您在叢集中執行的內容而有所不同，但對擴展影響最大的區域包括 Kubernetes 版本、使用率和個別節點擴展。

您可以使用兩種模式之一來執行叢集的控制平面，以滿足不同的工作負載需求：

- 標準模式 – 根據預設，所有 EKS 叢集都會使用標準模式。控制平面會根據工作負載需求自動擴展和縮減規模。標準模式可動態配置足夠的控制平面容量，是大多數使用案例的建議選項。
- 佈建模式 – 如果您的工作負載無法容忍控制平面擴展的效能變化，或者如果它們需要非常高的控制平面容量，您可以使用佈建模式。使用佈建模式，您可以預先配置控制平面容量，隨時準備好處理嚴苛的需求。您可以取得一致且可預測的效能。

使用 EKS 佈建模式時，您可以選擇一組擴展層 (XL、2XL、4XL 和 8XL)。透過每個層，您可以從叢集的控制平面獲得高、可預測的效能。佈建模式對於下列使用案例特別有用：

- 效能關鍵工作負載
- 大規模 AI 和機器學習操作
- 預期的高需求事件
- 需要跨預備和生產一致性的環境

透過佈建模式，您可以提前預先配置控制平面容量，並受益於以 1 分鐘間隔測量的增強型 99.99% 服務水準協議 (SLA)。如需 EKS 佈建模式的詳細資訊，包括方案規格和定價，請參閱《EKS 使用者指南》中的 EKS 佈建控制平面。

限制工作負載和節點爆量

> **Important**
>
> 為了避免在控制平面上達到 API 限制，您應該限制一次將叢集大小增加兩位數百分比的擴展峰值（例如，1000 個節點到 1100 個節點，或一次 4000 到 4500 個 Pod)。

EKS 控制平面會隨著叢集的成長而自動擴展，但擴展速度有所限制。當您第一次建立 EKS 叢集時，控制平面將無法立即擴展到數百個節點或數千個 Pod。若要進一步了解 EKS 如何改善擴展，請參閱此部落格文章。

擴展大型應用程式需要基礎設施才能適應完全就緒（例如暖機負載平衡器）。若要控制擴展速度，請確定您是根據應用程式的正確指標進行擴展。CPU 和記憶體擴展可能無法準確預測您的應用程式限制，並且在 Kubernetes Horizontal Pod Autoscaler (HPA) 中使用自訂指標（例如每秒請求）可能是更好的擴展選項。

若要使用自訂指標，請參閱 Kubernetes 文件中的範例。如果您有更進階的擴展需求，或需要根據外部來源（例如 AWS SQS 佇列）進行擴展，請使用 KEDA 進行事件型工作負載擴展。

安全地擴展節點和 Pod

取代長時間執行的執行個體

定期取代節點可避免組態偏離，以及僅在延長運作時間後才發生的問題（例如記憶體流失緩慢），以維持叢集的運作狀態。自動化替換將為您提供節點升級和安全修補的良好程序和實務。如果定期替換叢集中的每個節點，則維護個別程序以進行持續維護所需的時間較少。

使用 Karpenter 的存留時間 (TTL) 設定，在執行個體執行一段時間後取代執行個體。自我管理節點群組可以使用 `max-instance-lifetime`設定自動循環節點。受管節點群組目前沒有此功能，但您可以在 GitHub 上在此處追蹤請求。

移除未充分利用的節點

當節點沒有執行中的工作負載時，您可以使用 Kubernetes Cluster Autoscaler 中的縮減閾值搭配 `--scale-down-utilization-threshold`或在 Karpenter 中，來移除節點。您可以使用`ttlSecondsAfterEmpty`佈建器設定。

使用 Pod 中斷預算和安全節點關閉

從 Kubernetes 叢集移除 Pod 和節點需要控制器更新多個資源（例如 EndpointSlices)。頻繁或太快地執行此操作可能會導致 API 伺服器限流和應用程式中斷，因為變更會傳播到控制器。Pod 中斷預算是降低流失速度的最佳實務，以保護在叢集中移除或重新排程節點時的工作負載可用性。

執行 Kubectl 時使用用戶端快取

無效率地使用 kubectl 命令可新增額外負載至 Kubernetes API 伺服器。您應該避免重複執行使用 kubectl 的指令碼或自動化（例如在 的迴圈中），或在沒有本機快取的情況下執行命令。

`kubectl` 具有用戶端快取，可從叢集快取探索資訊，以減少所需的 API 呼叫數量。快取預設為啟用，每 10 分鐘重新整理一次。

如果您從容器執行 kubectl 或沒有用戶端快取，您可能會遇到 API 限流問題。建議您透過掛載 來保留叢集快取`--cache-dir`，以避免進行不必要的 API 呼叫。

停用 kubectl 壓縮

在 kubeconfig 檔案中停用 kubectl 壓縮可減少 API 和用戶端 CPU 使用量。根據預設，伺服器會壓縮傳送至用戶端的資料，以最佳化網路頻寬。這會為每個請求在用戶端和伺服器上新增 CPU 負載，而且如果您有足夠的頻寬，停用壓縮可以減少額外負荷和延遲。若要停用壓縮，您可以使用 `--disable-compression=true`旗標或在 `disable-compression: true` kubeconfig 檔案中設定 。

```
apiVersion: v1
clusters:
- cluster:
    server: serverURL
    disable-compression: true
  name: cluster
```

碎片叢集自動擴展器

Kubernetes Cluster Autoscaler 已經過測試，最多可擴展 1000 個節點。在節點超過 1000 個的大型叢集上，建議在碎片模式中執行 Cluster Autoscaler 的多個執行個體。每個 Cluster Autoscaler 執行個體都設定為擴展一組節點群組。下列範例顯示設定為每個擴展 4 節點群組的 2 個叢集自動擴展組態。

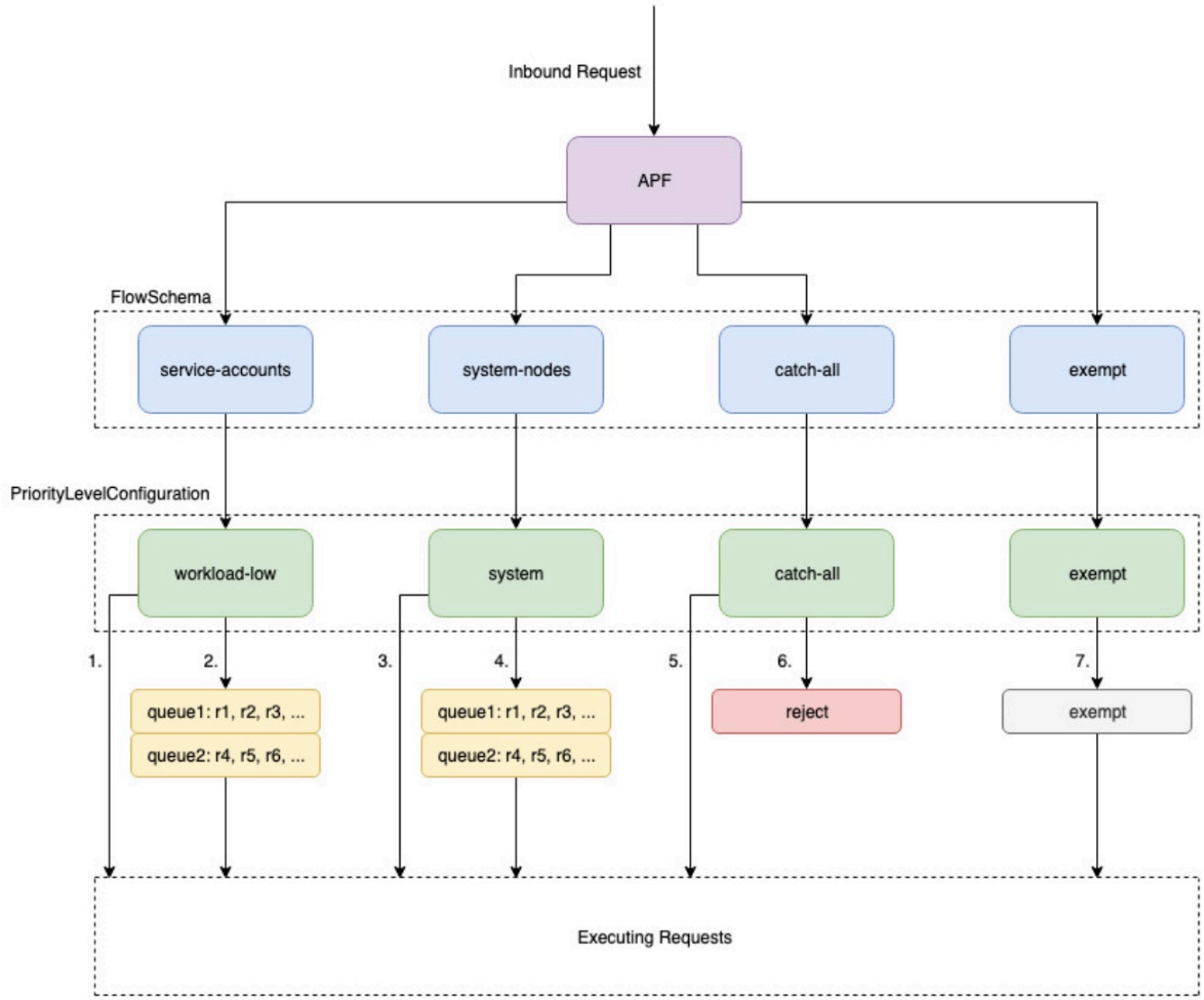
ClusterAutoscaler-1

```
autoscalingGroups:
- name: eks-core-node-grp-20220823190924690000000011-80c1660e-030d-476d-cb0d-
d04d585a8fcb
  maxSize: 50
  minSize: 2
- name: eks-data_m1-20220824130553925600000011-5ec167fa-ca93-8ca4-53a5-003e1ed8d306
  maxSize: 450
  minSize: 2
- name: eks-data_m2-20220824130733258600000015-aac167fb-8bf7-429d-d032-e195af4e25f5
  maxSize: 450
  minSize: 2
- name: eks-data_m3-20220824130553914900000003-18c167fa-ca7f-23c9-0fea-f9edefbda002
  maxSize: 450
  minSize: 2
```

ClusterAutoscaler-2

```
autoscalingGroups:
- name: eks-data_m4-20220824130553925500000000f-5ec167fa-ca86-6b83-ae9d-1e07ade3e7c4
  maxSize: 450
  minSize: 2
- name: eks-data_m5-20220824130744542100000017-02c167fb-a1f7-3d9e-a583-43b4975c050c
  maxSize: 450
  minSize: 2
- name: eks-data_m6-20220824130553924300000000d-9cc167fa-ca94-132a-04ad-e43166cef41f
  maxSize: 450
  minSize: 2
- name: eks-data_m7-20220824130553921000000009-96c167fa-ca91-d767-0427-91c879ddf5af
  maxSize: 450
  minSize: 2
```

API 優先順序和公平性



1. If workload-low has not reached its concurrency limit, the request will be dispatched immediately.
2. If workload-low has reached its concurrency limit, the request will be queued prior to dispatch.
3. If system has not reached its concurrency limit, the request will be dispatched immediately.
4. If system has reached its concurrency limit, the request will be queued prior to dispatch.
5. If catch-all has not reached its concurrency limit, the request will be dispatched immediately.
6. If catch-all has reached its concurrency limit, the request will be dropped.
7. Requests mapping to exempt will always be dispatched.

概觀

為了避免在請求增加期間超載，API Server 會限制其在特定時間可以有未完成的傳輸中請求數量。一旦超過此限制，API Server 將開始拒絕請求，並將「太多請求」的 429 HTTP 回應碼傳回給用戶端。

伺服器捨棄請求並在稍後讓用戶端重試，最好對請求數量和控制平面超載沒有伺服器端限制，這可能會導致效能降低或無法使用。

Kubernetes 用來設定這些傳輸中請求如何分成不同請求類型的機制稱為 API Priority and Fairness。API Server 透過加總 `--max-requests-inflight`和 `--max-mutating-requests-inflight`旗標指定的值，設定其可接受之傳輸中請求的總數。EKS 會針對這些旗標使用 400 和 200 個請求的預設值，允許在指定時間總共傳送 600 個請求。不過，當它將控制平面擴展到更大的大小以回應增加的使用率和工作負載流失時，它會相應地一直增加傳輸中請求配額，直到 2000 年（可能會有所變更）。APF 指定如何在不同的請求類型之間進一步細分這些傳輸中請求配額。請注意，EKS 控制平面具有至少 2 個 API 伺服器註冊到每個叢集的高可用性。這表示您的叢集可以處理的傳輸中請求總數是每個 kube-apiserver 的傳輸中配額集的兩倍（如果水平擴展，則更高）。這相當於最大 EKS 叢集上每秒數千個請求。

兩種類型的 Kubernetes 物件，稱為 PriorityLevelConfigurations 和 FlowSchemas，可設定在不同請求類型之間分割請求總數的方式。這些物件由 API Server 自動維護，EKS 會針對指定的 Kubernetes 次要版本使用這些物件的預設組態。PriorityLevelConfigurations 代表允許請求總數的一小部分。例如，高工作負載 PriorityLevelConfiguration 在總共 600 個請求中配置 98 個。分配給所有 PriorityLevelConfigurations 的請求總和將等於 600（或略高於 600，因為如果授予給給指定層級的一小部分請求，則 API 伺服器會四捨五入）。若要檢查叢集中的 PriorityLevelConfigurations，以及分配給每個叢集的請求數量，您可以執行下列命令。以下是 EKS 1.32 的預設值：

```
$ kubectl get --raw /metrics | grep apiserver_flowcontrol_nominal_limit_seats
apiserver_flowcontrol_nominal_limit_seats{priority_level="catch-all"} 13
apiserver_flowcontrol_nominal_limit_seats{priority_level="exempt"} 0
apiserver_flowcontrol_nominal_limit_seats{priority_level="global-default"} 49
apiserver_flowcontrol_nominal_limit_seats{priority_level="leader-election"} 25
apiserver_flowcontrol_nominal_limit_seats{priority_level="node-high"} 98
apiserver_flowcontrol_nominal_limit_seats{priority_level="system"} 74
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-high"} 98
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-low"} 245
```

第二種類型的物件是 FlowSchemas。具有指定屬性集的 API Server 請求會分類在相同的 FlowSchema 下。這些屬性包括已驗證的使用者或請求的屬性，例如 API 群組、命名空間或資源。FlowSchema 也會指定此類型請求應對應的 PriorityLevelConfiguration。這兩個物件一起說：「我想要這種類型的請求計入這種傳輸中請求的份額。」 當請求命中 API 伺服器時，它會檢查其每個 FlowSchemas，直到找到符合所有必要屬性的請求為止。如果多個 FlowSchemas 符合請求，API Server 將選擇具有最小相符優先順序的 FlowSchema，該優先順序指定為 物件中的屬性。

您可以使用此命令檢視 FlowSchemas 與 PriorityLevelConfigurations 的映射：

```
$ kubectl get flowschemas
NAME                           PRIORITYLEVEL     MATCHINGPRECEDENCE
 DISTINGUISHERMETHOD   AGE     MISSINGPL
exempt                         exempt            1                  <none>
     7h19m   False
eks-exempt                     exempt            2                  <none>
     7h19m   False
probes                         exempt            2                  <none>
     7h19m   False
system-leader-election         leader-election   100                ByUser
     7h19m   False
endpoint-controller            workload-high     150                ByUser
     7h19m   False
workload-leader-election       leader-election   200                ByUser
     7h19m   False
system-node-high               node-high         400                ByUser
     7h19m   False
system-nodes                   system            500                ByUser
     7h19m   False
kube-controller-manager        workload-high     800                ByNamespace
     7h19m   False
kube-scheduler                 workload-high     800                ByNamespace
     7h19m   False
kube-system-service-accounts   workload-high     900                ByNamespace
     7h19m   False
eks-workload-high              workload-high     1000               ByUser
     7h14m   False
service-accounts               workload-low      9000               ByUser
     7h19m   False
global-default                 global-default    9900               ByUser
     7h19m   False
catch-all                      catch-all         10000              ByUser
     7h19m   False
```

PriorityLevelConfigurations 可以有佇列、拒絕或豁免的類型。對於佇列和拒絕類型，會針對該優先順序層級的處理中請求數量上限強制執行限制，不過，達到該限制時，行為會有所不同。例如，高工作負載 PriorityLevelConfiguration 使用類型 Queue，並且有 98 個請求可供控制器管理員、端點控制器、排程器、eks 相關控制器以及在 kube-system 命名空間中執行的 Pod 使用。由於使用類型佇列，API Server 會嘗試將請求保留在記憶體中，並希望在這些請求逾時之前，處理中的請求數量低於 98。如果指定的請求在佇列中逾時，或已排入佇列的請求太多，API Server 就無法選擇捨棄請求，並傳回用戶端 429。請注意，佇列可能會阻止請求接收 429，但會因為請求的end-to-end延遲增加而遭到權衡。

現在，請考慮對應至具有類型 Reject 之 catch-all PriorityLevelConfiguration 的所有截獲 FlowSchema。如果用戶端達到 13 個傳輸中請求的限制，API Server 將不會執行佇列，並且會立即捨棄具有 429 回應碼的請求。最後，映射到類型為豁免的 PriorityLevelConfiguration 的請求永遠不會收到 429，而且一律會立即分派。這用於高優先順序的請求，例如 healthz 請求或來自 system：masters 群組的請求。

監控 APF 和捨棄的請求

若要確認是否因為 APF 而捨棄任何請求，`apiserver_flowcontrol_rejected_requests_total`可以監控 的 API 伺服器指標，以檢查受影響的 FlowSchemas 和 PriorityLevelConfigurations。例如，此指標顯示 100 個來自服務帳戶 FlowSchema 的請求因低工作負載佇列中的請求逾時而遭到捨棄：

```
% kubectl get --raw /metrics | grep apiserver_flowcontrol_rejected_requests_total
apiserver_flowcontrol_rejected_requests_total{flow_schema="service-
accounts",priority_level="workload-low",reason="time-out"} 100
```

若要檢查指定的 PriorityLevelConfiguration 與接收 429 秒的接近程度，或因佇列而經歷增加的延遲，您可以比較並行限制與使用中的並行之間的差異。在此範例中，我們有 100 個請求的緩衝區。

```
% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_nominal_limit_seats.*workload-low'
apiserver_flowcontrol_nominal_limit_seats{priority_level="workload-low"} 245

% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_current_executing_seats.*workload-low'
apiserver_flowcontrol_current_executing_seats{flow_schema="service-
accounts",priority_level="workload-low"} 145
```

若要檢查指定的 PriorityLevelConfiguration 是否遇到佇列，但不一定捨棄請求，`apiserver_flowcontrol_current_inqueue_requests`可以參考 的 指標：

```
% kubectl get --raw /metrics | grep
 'apiserver_flowcontrol_current_inqueue_requests.*workload-low'
apiserver_flowcontrol_current_inqueue_requests{flow_schema="service-
accounts",priority_level="workload-low"} 10
```

其他有用的 Prometheus 指標包括：

- apiserver_flowcontrol_dispatched_requests_total

- apiserver_flowcontrol_request_execution_seconds
- apiserver_flowcontrol_request_wait_duration_seconds

如需 APF 指標的完整清單，請參閱上游文件。

防止捨棄的請求

變更工作負載以防止 429

當 APF 因指定的 PriorityLevelConfiguration 而捨棄請求超過其允許的最大傳輸中請求數時，受影響 FlowSchemas 中的用戶端可以減少在指定時間執行的請求數。這可以透過減少在 429 個期間內提出的請求總數來完成。請注意，長時間執行的請求，例如昂貴的清單呼叫，特別有問題，因為它們在執行的整個期間都算是傳輸中請求。減少這些昂貴請求的數量或最佳化這些清單呼叫的延遲（例如，透過減少每個請求擷取的物件數量或使用監看請求切換到），有助於減少指定工作負載所需的並行總數。

變更您的 APF 設定以防止 429

> **Warning**
>
> 只有在您知道正在做什麼時，才能變更預設 APF 設定。設定錯誤的 APF 設定可能會導致 API 伺服器請求遭到捨棄，並造成嚴重的工作負載中斷。

防止捨棄請求的另一種方法是變更 EKS 叢集上安裝的預設 FlowSchemas 或 PriorityLevelConfigurations。EKS 會為指定 Kubernetes 次要版本安裝 FlowSchemas 和 PriorityLevelConfigurations 的上游預設設定。除非物件上的下列註釋設定為 false，否則 API Server 會自動將這些物件調整回其預設值：

```
metadata:
  annotations:
    apf.kubernetes.io/autoupdate-spec: "false"
```

在高階，APF 設定可以修改為：

- 將更多傳輸中容量分配給您關心的請求。
- 隔離可能讓其他請求類型容量不足的非必要或昂貴請求。

這可以透過變更預設 FlowSchemas 和 PriorityLevelConfigurations 或建立這些類型的新物件來完成。運算子可以增加相關 PriorityLevelConfigurations 物件的 assuredConcurrencyShares 值，以增加配置

的傳輸中請求部分。此外，如果應用程式可以處理在發出請求之前排入佇列而導致的額外延遲，則可以增加可在指定時間排入佇列的請求數量。

或者，您可以建立客戶工作負載特有的新 FlowSchema 和 PriorityLevelConfigurations 物件。請注意，將更多 assuredConcurrencyShares 配置到現有的 PriorityLevelConfigurations 或新的 PriorityLevelConfigurations 將導致其他儲存貯體可以處理的請求數量減少，因為整體限制將保留為每個 API 伺服器 600 個傳輸中。

變更 APF 預設值時，應在非生產叢集上監控這些指標，以確保變更設定不會導致意外的 429s：

1. `apiserver_flowcontrol_rejected_requests_total` 應監控所有 FlowSchemas 的 指標，以確保沒有任何儲存貯體開始捨棄請求。
2. `apiserver_flowcontrol_current_executing_seats` 應比較 `apiserver_flowcontrol_nominal_limit_seats`和 的值，以確保使用中的並行不會違反該優先順序層級的限制。

定義新 FlowSchema 和 PriorityLevelConfiguration 的一個常見使用案例是用於隔離。假設我們想要隔離長期執行的清單事件呼叫，從 Pod 到他們自己的請求份額。這將防止來自使用現有服務帳戶 FlowSchema 的 Pod 的重要請求接收 429s 並被請求容量綁定。請記住，傳輸中請求的總數有限，不過，此範例顯示可以修改 APF 設定，以便更好地分割指定工作負載的請求容量：

隔離清單事件請求的 FlowSchema 物件範例：

```
apiVersion: flowcontrol.apiserver.k8s.io/v1
kind: FlowSchema
metadata:
  name: list-events-default-service-accounts
spec:
  distinguisherMethod:
    type: ByUser
  matchingPrecedence: 8000
  priorityLevelConfiguration:
    name: catch-all
  rules:
  - resourceRules:
    - apiGroups:
      - '*'
      namespaces:
      - default
      resources:
```

```
    - events
    verbs:
    - list
  subjects:
  - kind: ServiceAccount
    serviceAccount:
      name: default
      namespace: default
```

- 此 FlowSchema 會擷取服務帳戶在預設命名空間中進行的所有事件呼叫清單。
- 比對優先順序 8000 低於現有服務帳戶 FlowSchema 所使用的 9000 值，因此這些清單事件呼叫將比對 list-events-default-service-accounts，而不是 service-accounts。
- 我們使用全部的 PriorityLevelConfiguration 來隔離這些請求。此儲存貯體僅允許這些長時間執行的清單事件呼叫使用 13 個傳輸中請求。一旦 Pod 嘗試同時發出超過 13 個請求，就會開始收到 429 個請求。

擷取 API 伺服器中的資源

從 API 伺服器取得資訊是任何大小叢集的預期行為。隨著您擴展叢集中的資源數量，請求頻率和資料量可能會很快成為控制平面的瓶頸，並導致 API 延遲和速度變慢。根據延遲的嚴重性，如果不小心，會導致非預期的停機時間。

了解您的請求，以及避免這類問題的第一步有多頻繁。以下是根據擴展最佳實務限制查詢數量的指引。本節中的建議會從已知可擴展最佳選項開始依序提供。

使用共用資訊提供者

建置與 Kubernetes API 整合的控制器和自動化時，您通常需要從 Kubernetes 資源取得資訊。如果您定期輪詢這些資源，可能會對 API 伺服器造成大量負載。

使用來自 client-go 程式庫的資訊提供者，可讓您根據事件觀看資源的變更，而不是輪詢變更。資訊提供者會使用事件和變更的共用快取進一步減少負載，因此多個監看相同資源的控制器不會新增額外的負載。

控制器應避免在沒有標籤和欄位選擇器的情況下輪詢整個叢集的資源，尤其是在大型叢集中。每個未篩選的輪詢都需要用戶端透過 API 伺服器從 etcd 傳送許多不必要的資料。透過根據標籤和命名空間進行篩選，您可以減少 API 伺服器需要執行的工作量，以完整記錄傳送至用戶端的請求和資料。

最佳化 Kubernetes API 用量

使用自訂控制器或自動化呼叫 Kubernetes API 時，請務必將呼叫限制為僅需要的資源。如果沒有限制，可能會在 API 伺服器和 等上造成不需要的載入。

建議您盡可能使用 Watch 引數。沒有引數時，預設行為是列出物件。若要使用監看而非清單，您可以將 附加`?watch=true`到 API 請求的結尾。例如，若要使用監看功能取得預設命名空間中的所有 Pod：

```
/api/v1/namespaces/default/pods?watch=true
```

如果您要列出物件，您應該限制要列出的範圍和傳回的資料量。您可以透過將`limit=500`引數新增至請求來限制傳回的資料。`fieldSelector` 引數和`/namespace/`路徑對於確保您的清單視需要縮小範圍很有用。例如，若要僅列出預設命名空間中執行中的 Pod，請使用下列 API 路徑和引數。

```
/api/v1/namespaces/default/pods?fieldSelector=status.phase=Running&limit=500
```

或列出所有執行中的 Pod：

```
/api/v1/pods?fieldSelector=status.phase=Running&limit=500
```

限制監看呼叫或列出的物件的另一個選項是使用 `resourceVersions`，您可以在 Kubernetes 文件中閱讀這些內容。如果沒有 `resourceVersion` 引數，您將會收到可用的最新版本，該版本需要對資料庫而言最昂貴且最慢的刻板讀取。resourceVersion 取決於您嘗試查詢的資源，可在 `metadata.resourseVersion` 欄位中找到。如果使用監看呼叫，而不是只列出呼叫，也建議這樣做

有一個特殊`resourceVersion=0`的可用項目，將從 API 伺服器快取傳回結果。這可以減少刻印負載，但不支援分頁。

```
/api/v1/namespaces/default/pods?resourceVersion=0
```

建議將具有 resourceVersion 的 watch 設定為從上一個清單或 watch 收到的最新已知值。這會在 client-go 中自動處理。但是，如果您使用其他語言的 k8s 用戶端，建議您再次檢查。

```
/api/v1/namespaces/default/pods?watch=true&resourceVersion=362812295
```

如果您在沒有任何引數的情況下呼叫 API，將會是 API 伺服器和 等的最密集資源。此呼叫將取得所有命名空間中的所有 Pod，而不會分頁或限制範圍，並且需要從 等方讀取規定人數。

```
/api/v1/pods
```

防止 DaemonSet 雷擊雜草

DaemonSet 可確保所有（或部分）節點執行 Pod 的副本。當節點加入叢集時，daemonset-controller 會為這些節點建立 Pod。當節點離開叢集時，會收集這些 Pod。刪除 DaemonSet 會清除其建立的 Pod。

DaemonSet 的一些典型用途如下：

- 在每個節點上執行叢集儲存協助程式
- 在每個節點上執行日誌集合常駐程式
- 在每個節點上執行節點監控協助程式

在具有數千個節點的叢集上，建立新的 DaemonSet、更新 DaemonSet 或增加節點數量可能會導致在控制平面上放置高負載。如果 DaemonSet Pod 在 Pod 啟動時發出昂貴的 API 伺服器請求，可能會導致大量並行請求在控制平面上使用大量資源。

在正常操作中，您可以使用 `RollingUpdate`來確保逐步推出新的 DaemonSet Pod。透過`RollingUpdate`更新策略，在您更新 DaemonSet 範本之後，控制器會終止舊的 DaemonSet Pod，並以受控方式自動建立新的 DaemonSet Pod。在整個更新過程中，每個節點上最多都會執行一個 DaemonSet Pod。您可以將 `maxUnavailable`設定為 1、`maxSurge`0 和 `minReadySeconds` 60，以執行逐步推展。如果您未指定更新策略，Kubernetes 會將 預設為以 `RollingUpdate` `maxUnavailable` 1、0 `maxSurge` 和 0 `minReadySeconds`建立 。

```
minReadySeconds: 60
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 0
    maxUnavailable: 1
```

如果已建立 DaemonSet 且在所有節點上具有預期的 Pod 數量，則 `RollingUpdate`可確保逐步推出新的 DaemonSet `Ready` Pod。在某些情況下，可能會在`RollingUpdate`策略未涵蓋的特定條件下產生雜湊問題。

防止在 DaemonSet 建立時產生雜湊雜湊

根據預設，無論RollingUpdate組態為何，kube-controller-manager 中的 daemonset-controller 都會在您建立新的 DaemonSet 時，同時為所有相符節點建立 Pod。若要在建立 DaemonSet 後強制逐步推出 Pod，您可以使用 NodeSelector或 NodeAffinity。這將建立符合零節點的 DaemonSet，然後您可以逐步更新節點，使其有資格以受控速率從 DaemonSet 執行 Pod。您可以遵循以下方法：

- 將標籤新增至 的所有節點run-daemonset=false。

```
kubectl label nodes --all run-daemonset=false
```

- 使用 NodeAffinity設定建立您的 DaemonSet，以符合沒有run-daemonset=false標籤的任何節點。一開始，這將導致您的 DaemonSet 沒有對應的 Pod。

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: run-daemonset
          operator: NotIn
          values:
          - "false"
```

- 以受控速率從節點移除run-daemonset=false標籤。您可以使用此 bash 指令碼做為範例：

```
#!/bin/bash

nodes=$(kubectl get --raw "/api/v1/nodes" | jq -r '.items | .[].metadata.name')

for node in ${nodes[@]}; do
   echo "Removing run-daemonset label from node $node"
   kubectl label nodes $node run-daemonset-
   sleep 5
done
```

- 或者，從 DaemonSet 物件中移除 NodeAffinity設定。請注意，這也會觸發 ，RollingUpdate並逐漸取代所有現有的 DaemonSet Pod，因為 DaemonSet 範本已變更。

防止節點橫向擴展上的雜湊雜湊

與 DaemonSet 建立類似，快速建立新節點可能會導致大量 DaemonSet Pod 同時啟動。您應該以受控速率建立新的節點，讓控制器以相同的速率建立 DaemonSet Pod。如果無法這麼做，您可以使用 讓新節點最初不符合現有 DaemonSet 的資格`NodeAffinity`。接下來，您可以逐步將標籤新增至新節點，讓 daemonset-controller 以受控速率建立 Pod。您可以遵循以下方法：

- 將標籤新增至 的所有現有節點 `run-daemonset=true`

```
kubectl label nodes --all run-daemonset=true
```

- 使用 `NodeAffinity`設定更新您的 DaemonSet，以符合任何具有`run-daemonset=true`標籤的節點。請注意，這也會觸發 ，`RollingUpdate`並逐漸取代所有現有的 DaemonSet Pod，因為 DaemonSet 範本已變更。在進入下一個步驟之前`RollingUpdate`，您應該等待 完成。

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: run-daemonset
          operator: In
          values:
          - "true"
```

- 在叢集中建立新的節點。請注意，這些節點不會有 `run-daemonset=true` 標籤，因此 DaemonSet 不會符合這些節點。
- 以受控速率將`run-daemonset=true`標籤新增至新節點 （目前沒有`run-daemonset`標籤）。您可以使用此 bash 指令碼做為範例：

```
#!/bin/bash

nodes=$(kubectl get --raw "/api/v1/nodes?labelSelector=%21run-daemonset" | jq -r
 '.items | .[].metadata.name')

for node in ${nodes[@]}; do
   echo "Adding run-daemonset=true label to node $node"
```

```
   kubectl label nodes $node run-daemonset=true
   sleep 5
done
```

- 或者，從 DaemonSet 物件移除NodeAffinity設定，並從所有節點移除run-daemonset標籤。

防止 DaemonSet 更新上的雜湊雜湊

RollingUpdate 政策只會遵守 之 DaemonSet Pod maxUnavailable的設定Ready。如果 DaemonSet NotReady 只有 Pod NotReady 或大量 Pod，且您更新其範本，則 daemonset-controller 會同時為任何 Pod 建立新的 NotReady Pod。如果 Pod NotReady 數量很大，例如 Pod 持續當機循環或無法提取映像，這可能會導致雷擊問題。

若要在更新 DaemonSet 且有 Pod 時強制逐步推出 NotReady Pod，您可以暫時將 DaemonSet 上的更新策略從 變更為 RollingUpdate OnDelete。使用 OnDelete更新 DaemonSet 範本後，控制器會在您手動刪除舊的 Pod 之後建立新的 Pod，讓您可以控制新 Pod 的推出。您可以遵循以下方法：

- 檢查您的 DaemonSet NotReady 中是否有任何 Pod。
- 如果否，您可以安全地更新 DaemonSet 範本，且RollingUpdate策略將確保逐步推出。
- 如果是，您應該先更新 DaemonSet 以使用 OnDelete策略。

```
updateStrategy:
  type: OnDelete
```

- 接著，使用所需的變更更新您的 DaemonSet 範本。
- 在此更新之後，您可以透過以受控速率發出刪除 Pod 請求來刪除舊的 DaemonSet Pod。您可以使用此 bash 指令碼做為範例，其中 DaemonSet 名稱在 kube-system 命名空間中為 fluentd-elasticsearch：

```
#!/bin/bash

daemonset_pods=$(kubectl get --raw "/api/v1/namespaces/kube-system/pods?
labelSelector=name%3Dfluentd-elasticsearch" | jq -r '.items | .[].metadata.name')

for pod in ${daemonset_pods[@]}; do
   echo "Deleting pod $pod"
   kubectl delete pod $pod -n kube-system
```

```
    sleep 5
done
```

- 最後，您可以將 DaemonSet 更新回先前的`RollingUpdate`策略。

Kubernetes 資料平面

選取 EC2 執行個體類型可能是客戶面臨的最困難決策之一，因為在具有多個工作負載的叢集中。沒有 one-size-fits解決方案。以下是一些秘訣，可協助您避免擴展運算的常見陷阱。

自動節點自動擴展

我們建議您使用節點自動調整規模，以減少混音並與 Kubernetes 深度整合。建議大規模叢集使用受管節點群組和 Karpenter。

受管節點群組將為您提供 Amazon EC2 Auto Scaling 群組的彈性，並新增受管升級和組態的優點。它可以使用 Kubernetes Cluster Autoscaler 進行擴展，並且是具有各種運算需求的叢集的常見選項。

Karpenter 是由 AWS 建立的開放原始碼、工作負載原生節點自動擴展器。它會根據資源 （ 例如 GPU) 的工作負載需求，以及污點和容錯 （ 例如區域分散 ） 來擴展叢集中的節點，而無需管理節點群組。節點直接從 EC2 建立，可避免預設節點群組配額 - 每個群組 450 個節點 - 並提供更大的執行個體選擇彈性，並降低營運開銷。我們建議客戶盡可能使用 Karpenter。

使用許多不同的 EC2 執行個體類型

每個 AWS 區域每個執行個體類型的可用執行個體數量有限。如果您建立的叢集只使用一種執行個體類型，並將節點數量擴展到超過區域的容量，您會收到沒有執行個體可用的錯誤。為了避免此問題，您不應任意限制叢集中可以使用的執行個體類型。

Karpenter 預設將使用廣泛的相容執行個體類型，並根據待處理工作負載需求、可用性和成本在佈建時挑選執行個體。您可以擴大 NodePools `karpenter.k8s.aws/instance-category`金鑰中使用的執行個體類型清單。

Kubernetes Cluster Autoscaler 需要類似大小的節點群組，才能持續擴展。您應該根據 CPU 和記憶體大小建立多個群組，並獨立擴展。使用 ec2-instance-selector 來識別節點群組大小相似的執行個體。

```
ec2-instance-selector --service eks --vcpus-min 8 --memory-min 16
a1.2xlarge
a1.4xlarge
a1.metal
```

```
c4.4xlarge
c4.8xlarge
c5.12xlarge
c5.18xlarge
c5.24xlarge
c5.2xlarge
c5.4xlarge
c5.9xlarge
c5.metal
```

偏好較大的節點以減少 API 伺服器負載

決定要使用的執行個體類型時，較少的大型節點會對 Kubernetes 控制平面造成較少的負載，因為會有較少的 kubelet 和執行中的 DaemonSets。不過，大型節點可能無法完全使用，就像較小的節點一樣。節點大小應根據您的工作負載可用性和擴展需求進行評估。

具有三個 u-24tb1.metal 執行個體 (24 TB 記憶體和 448 個核心） 的叢集有 3 個 kubelet，預設每個節點限制為 110 個 Pod。如果您的 Pod 各使用 4 個核心，則可能預期會發生這種情況 (4 個核心 x 110 = 440 個核心/節點）。使用 3 節點叢集時，處理執行個體事件的能力會很低，因為 1 個執行個體中斷可能會影響叢集的 1/3。您應該在工作負載中指定節點需求和 Pod，以便 Kubernetes 排程器可以正確放置工作負載。

工作負載應定義所需的資源，以及透過污點、容錯和 PodTopologySpread 所需的可用性。他們應該偏好可充分利用的最大節點，並符合可用性目標，以降低控制平面負載、降低操作並降低成本。

如果資源可用，Kubernetes 排程器會自動嘗試將工作負載分散到可用區域和主機。如果沒有可用的容量，Kubernetes Cluster Autoscaler 會嘗試在每個可用區域中平均新增節點。除非工作負載指定其他需求，否則 Karpenter 會嘗試盡可能快速且經濟實惠地新增節點。

若要強制工作負載隨著排程器和新節點分散到可用區域，您應該使用 topologySpreadConstraints：

```
spec:
  topologySpreadConstraints:
    - maxSkew: 3
      topologyKey: "topology.kubernetes.io/zone"
      whenUnsatisfiable: ScheduleAnyway
      labelSelector:
        matchLabels:
          dev: my-deployment
    - maxSkew: 2
      topologyKey: "kubernetes.io/hostname"
      whenUnsatisfiable: ScheduleAnyway
```

```
      labelSelector:
        matchLabels:
          dev: my-deployment
```

使用類似的節點大小來取得一致的工作負載效能

工作負載應定義需要執行的節點大小，以允許一致的效能和可預測的擴展。請求 500 公尺 CPU 的工作負載在具有 4 個核心的執行個體上執行，與具有 16 個核心的執行個體執行方式不同。避免使用爆量 CPUs執行個體類型，例如 T 系列執行個體。

為了確保工作負載獲得一致的效能，工作負載可以使用支援的 Karpenter 標籤來鎖定特定執行個體大小。

```
kind: deployment
...
spec:
  template:
    spec:
    containers:
    nodeSelector:
      karpenter.k8s.aws/instance-size: 8xlarge
```

使用 Kubernetes Cluster Autoscaler 在叢集中排程的工作負載，應根據標籤比對將節點選擇器與節點群組進行比對。

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: eks.amazonaws.com/nodegroup
            operator: In
            values:
            - 8-core-node-group    # match your node group name
```

有效率地使用運算資源

運算資源包括 EC2 執行個體和可用區域。有效使用運算資源可提高您的可擴展性、可用性、效能，並降低您的總成本。在具有多個應用程式的自動擴展環境中，高效的資源使用非常困難。建立 Karpenter 是為了根據工作負載需求隨需佈建執行個體，以最大化使用率和彈性。

Karpenter 允許工作負載宣告其所需的運算資源類型，而無需先建立節點群組或設定特定節點的標籤污點。如需詳細資訊，請參閱 Karpenter 最佳實務。請考慮在 Karpenter 佈建器中啟用整合，以取代使用率不足的節點。

自動化 Amazon Machine Image (AMI) 更新

將工作者節點元件保持在最新狀態，可確保您擁有最新的安全修補程式和與 Kubernetes API 相容的功能。更新 kubelet 是 Kubernetes 功能最重要的元件，但自動化作業系統、核心和本機安裝的應用程式修補程式將減少您擴展時的維護。

建議您為節點映像使用最新的 Amazon EKS 最佳化 Amazon Linux 2 或 Amazon EKS 最佳化 Bottlerocket AMI。Karpenter 將自動使用最新的可用 AMI 來佈建叢集中的新節點。受管節點群組將在節點群組更新期間更新 AMI，但不會在節點佈建時間更新 AMI ID。

對於受管節點群組，您需要在可用於修補程式版本時，使用新的 AMI IDs 更新 Auto Scaling 群組 (ASG) 啟動範本。AMI 次要版本（例如 1.23.5 到 1.24.3) 將可在 EKS 主控台和 API 中作為節點群組的升級。修補程式發行版本（例如 1.23.5 到 1.23.6) 不會顯示為節點群組的升級。如果您想要讓節點群組與 AMI 修補程式版本保持最新狀態，您需要建立新的啟動範本版本，並讓節點群組將執行個體取代為新的 AMI 版本。

您可以從此頁面找到最新的可用 AMI，或使用 AWS CLI。

```
aws ssm get-parameter \
  --name /aws/service/eks/optimized-ami/1.24/amazon-linux-2/recommended/image_id \
  --query "Parameter.Value" \
  --output text
```

針對容器使用多個 EBS 磁碟區

EBS 磁碟區具有根據磁碟區類型（例如 gp3) 和磁碟大小的輸入/輸出 (I/O) 配額。如果您的應用程式與主機共用單一 EBS 根磁碟區，這可能會耗盡整個主機的磁碟配額，並導致其他應用程式等待可用容量。如果應用程式將檔案寫入其浮水印分割區、從主機掛載本機磁碟區，以及根據所使用的記錄代理程式登入標準輸出 (STDOUT)，則會寫入磁碟。

為了避免磁碟 I/O 耗盡，您應該將第二個磁碟區掛載到容器狀態資料夾（例如 /run/containerd)，請使用單獨的 EBS 磁碟區進行工作負載儲存，並停用不必要的本機記錄。

若要使用 eksctl 將第二個磁碟區掛載到您的 EC2 執行個體，您可以使用具有此組態的節點群組：

```
managedNodeGroups:
```

```
  - name: al2-workers
    amiFamily: AmazonLinux2
    desiredCapacity: 2
    volumeSize: 80
    additionalVolumes:
      - volumeName: '/dev/sdz'
        volumeSize: 100
    preBootstrapCommands:
    - |
      "systemctl stop containerd"
      "mkfs -t ext4 /dev/nvme1n1"
      "rm -rf /var/lib/containerd/*"
      "mount /dev/nvme1n1 /var/lib/containerd/"
      "systemctl start containerd"
```

如果您使用 terraform 來佈建節點群組，請參閱 EKS 藍圖中 terraform 的範例。如果您使用 Karpenter 佈建節點，則可以`blockDeviceMappings`搭配使用節點使用者資料來新增其他磁碟區。

若要將 EBS 磁碟區直接掛載到您的 Pod，您應該使用 AWS EBS CSI 驅動程式，並使用具有儲存類別的磁碟區。

```
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-sc
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-claim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
    requests:
      storage: 4Gi
---
apiVersion: v1
kind: Pod
```

```
metadata:
  name: app
spec:
  containers:
  - name: app
    image: public.ecr.aws/docker/library/nginx
    volumeMounts:
    - name: persistent-storage
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: ebs-claim
```

如果工作負載使用 EBS 磁碟區，請避免具有低 EBS 連接限制的執行個體

EBS 是工作負載擁有持久性儲存的最簡單方法之一，但也附帶可擴展性限制。每個執行個體類型都有可連接的 EBS 磁碟區數量上限。工作負載需要宣告應該在哪些執行個體類型上執行，並限制具有 Kubernetes 污點的單一執行個體上的複本數量。

停用對磁碟不必要的記錄

避免不必要的本機記錄，方法是不要在生產環境中使用偵錯記錄執行應用程式，並停用經常讀取和寫入磁碟的記錄。日誌記錄是本機日誌記錄服務，可將日誌緩衝區保留在記憶體中，並定期排清至磁碟。日誌優先於將每行立即記錄到磁碟的 syslog。停用 syslog 也會降低您需要的總儲存量，並避免需要複雜的日誌輪換規則。若要停用 syslog，您可以將下列程式碼片段新增至 cloud-init 組態：

```
runcmd:
  - [ systemctl, disable, --now, syslog.service ]
```

當作業系統更新速度為必要時，就有修補程式執行個體

> **Important**
>
> 只有在需要時才應完成就地修補執行個體。Amazon 建議將基礎設施視為不可變且徹底測試透過較低環境提升的更新，就像應用程式一樣。本節適用於無法這麼做的情況。

在現有的 Linux 主機上安裝套件需要幾秒鐘的時間，而不會中斷容器化工作負載。套件可以安裝和驗證，無需封鎖、耗盡或取代執行個體。

若要取代執行個體，您必須先建立、驗證和分發新的 AMIs。執行個體需要建立替換，且舊執行個體需要封鎖和耗盡。然後，必須在新執行個體上建立工作負載，並針對所有需要修補的執行個體進行驗證和重複。安全地取代執行個體需要數小時、數天或數週的時間，而不會中斷工作負載。

Amazon 建議使用從自動化宣告式系統建置、測試和提升的不可變基礎設施，但如果您需要快速修補系統，則需要修補系統，並在新的 AMIs 可用時予以取代。由於修補和取代系統之間的時間差異很大，我們建議您在需要時使用 AWS Systems Manager Patch Manager 自動化修補節點。

修補節點可讓您快速推出安全性更新，並在 AMI 更新後定期取代執行個體。如果您使用作業系統搭配唯讀根檔案系統，例如 Flatcar Container Linux 或 Bottlerocket OS，我們建議您使用可搭配這些作業系統使用的更新運算子。Flatcar Linux 更新運算子和 Bottlerocket 更新運算子會重新啟動執行個體，讓節點自動保持最新狀態。

叢集服務

叢集服務會在 EKS 叢集內執行，但不是使用者工作負載。如果您有 Linux 伺服器，通常需要執行 NTP、syslog 和容器執行時間等服務來支援工作負載。叢集服務類似，支援服務可協助您自動化和操作叢集。在 Kubernetes 中，這些通常會在 kube-system 命名空間中執行，有些會以 DaemonSets 的形式執行。

叢集服務預期具有較高的運作時間，通常在中斷和故障診斷期間至關重要。如果核心叢集服務無法使用，您可能會無法存取有助於復原或防止中斷的資料（例如高磁碟使用率）。它們應該在專用運算執行個體上執行，例如單獨的節點群組或 AWS Fargate。這將確保叢集服務不會受到可能擴展或使用更多資源的工作負載對共用執行個體的影響。

擴展 CoreDNS

擴展 CoreDNS 有兩個主要機制。減少對 CoreDNS 服務的呼叫數量，並增加複本數量。

透過降低點陣來減少外部查詢

ndots 設定會指定網域名稱中有多少個句點 (a.k.a. "dots") 被視為足以避免查詢 DNS。如果您的應用程式有 5（預設）的 ndots 設定，而且您從外部網域請求資源，例如 api.example.com (2 個點），則會針對更特定網域在 /etc/resolv.conf 中定義的每個搜尋網域查詢 CoreDNS。根據預設，系統會在發出外部請求之前搜尋下列網域。

```
api.example.<namespace>.svc.cluster.local
api.example.svc.cluster.local
api.example.cluster.local
```

```
api.example.<region>.compute.internal
```

namespace 和 region值將取代為您的工作負載命名空間和運算區域。根據您的叢集設定，您可能還有其他搜尋網域。

您可以降低工作負載的 ndots 選項，或包含結尾（例如 api.example.com.）來完全限定您的網域請求，以減少對 CoreDNS 的請求數量。如果您的工作負載透過 DNS 連線至外部服務，建議您將 ndot 設定為 2，讓工作負載不會在叢集內進行不必要的叢集 DNS 查詢。如果工作負載不需要存取叢集內的服務，您可以設定不同的 DNS 伺服器和搜尋網域。

```
spec:
  dnsPolicy: "None"
  dnsConfig:
    options:
      - name: ndots
        value: "2"
      - name: edns0
```

如果您將點降低到太低的值，或者您連線的網域未包含足夠的特異性（包括結尾 。），則 DNS 查詢可能會失敗。請務必測試此設定將如何影響您的工作負載。

水平擴展 CoreDNS

CoreDNS 執行個體可以透過將其他複本新增至部署來擴展。建議您使用 NodeLocal DNS 或叢集比例自動擴展器來擴展 CoreDNS。

NodeLocal DNS 需要每個節點執行一個執行個體，做為 DaemonSet，這需要叢集中的更多運算資源，但可以避免 DNS 請求失敗，並減少叢集中 DNS 查詢的回應時間。叢集比例自動擴展器會根據叢集中的節點或核心數量來擴展 CoreDNS。這與請求查詢沒有直接關聯，但根據您的工作負載和叢集大小可能很有用。預設比例擴展是為叢集中每 256 個核心或 16 個節點新增一個額外的複本，以先發生者為準。

如果使用 CoreDNS EKS 附加元件，請考慮啟用自動調整規模選項。CoreDNS Autoscaler 會使用 (nodes÷16) 或 (CPU cores÷256) 的公式，視需要立即向上和向下擴展，以逐漸監控節點計數和 CPU 核心，動態調整 CoreDNS 複本的數量。

垂直擴展 Kubernetes 指標伺服器

Kubernetes Metrics Server 支援水平和垂直擴展。透過水平擴展指標伺服器，它將具有高可用性，但不會水平擴展以處理更多叢集指標。當節點和收集的指標新增至叢集時，您將需要根據指標伺服器的建議垂直擴展指標伺服器。

Metrics Server 會保留其在記憶體中收集、彙總和提供的資料。隨著叢集的成長，指標伺服器存放的資料量會增加。在大型叢集中，相較於預設安裝中指定的記憶體和 CPU 保留，Metrics Server 需要更多的運算資源。您可以使用 Vertical Pod Autoscaler (VPA) 或 Addon Resizer 來擴展指標伺服器。附加元件恢復器會根據工作者節點按比例垂直擴展，而 VPA 會根據 CPU 和記憶體用量擴展。

CoreDNS lameduck 持續時間

Pod 使用 kube-dns服務進行名稱解析。Kubernetes 使用目的地 NAT (DNAT) 將kube-dns流量從節點重新導向至 CoreDNS 後端 Pod。當您擴展 CoreDNS 部署時， 會kube-proxy更新節點上的 iptable 規則和鏈結，將 DNS 流量重新導向至 CoreDNS Pod。當您擴展時傳播新端點，並在縮減 CoreDNS 時刪除規則，可能需要 1 到 10 秒，取決於叢集的大小。

當 CoreDNS Pod 終止，但節點的 iptables 規則尚未更新時，此傳播延遲可能會導致 DNS 查詢失敗。在此案例中，節點可能會繼續將 DNS 查詢傳送至已終止的 CoreDNS Pod。

您可以在 CoreDNS Pod 中設定 lameduck 持續時間，以減少 DNS 查詢失敗。在 lameduck 模式下，CoreDNS 將繼續回應傳輸中的請求。設定 lameduck 持續時間將延遲 CoreDNS 關閉程序，讓節點有更新其 iptables 規則和鏈結所需的時間。

我們建議將 CoreDNS lameduck 持續時間設定為 30 秒。

CoreDNS 準備程度探查

我們建議您使用 /ready而非 /health進行 CoreDNS 的整備探查。

與先前的建議一致，將層壓持續時間設定為 30 秒，為節點的 iptables 規則提供足夠的時間，以便在 Pod 終止之前更新，採用 /ready 而不是 /health CoreDNS 整備探查，可確保 CoreDNS Pod 在啟動時完全準備好，以快速回應 DNS 請求。

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8181
    scheme: HTTP
```

如需 CoreDNS Ready 外掛程式的詳細資訊，請參閱 https：//https://coredns.io/plugins/ready/

記錄和監控代理程式

記錄和監控代理程式可以為您的叢集控制平面新增大量負載，因為代理程式會查詢 API 伺服器，以使用工作負載中繼資料豐富日誌和指標。節點上的代理程式只能存取本機節點資源，以查看容器和程序名

稱等物件。查詢 API 伺服器可以新增更多詳細資訊，例如 Kubernetes 部署名稱和標籤。這對於疑難排解非常有用，但對擴展不利。

由於記錄和監控有許多不同的選項，我們無法顯示每個供應商的範例。使用 fluentbit 時，我們建議啟用 Use_Kubelet 以從本機 kubelet 而非 Kubernetes API 伺服器擷取中繼資料，並將 `Kube_Meta_Cache_TTL`設定為可在資料快取時減少重複呼叫的數字（例如 60)。

擴展監控和記錄有兩個一般選項：

- 停用整合
- 取樣和篩選

停用整合通常不是選項，因為您遺失了日誌中繼資料。這樣可以消除 API 擴展問題，但它會在需要時沒有所需的中繼資料來引入其他問題。

抽樣和篩選可減少收集的指標和日誌數量。這將減少對 Kubernetes API 的請求量，並降低收集的指標和日誌所需的儲存量。降低儲存成本可降低整體系統的成本。

設定抽樣的能力取決於代理程式軟體，並且可以在不同的擷取點實作。請務必盡可能將抽樣新增至接近客服人員的位置，因為這可能是 API 伺服器呼叫發生的位置。請聯絡您的供應商，進一步了解抽樣支援。

如果您使用的是 CloudWatch 和 CloudWatch Logs，則可以使用文件中描述的模式來新增代理程式篩選。

為了避免遺失日誌和指標，您應該將資料傳送到可在接收端點發生中斷時緩衝資料的系統。透過 fluentbit，您可以使用 Amazon Kinesis Data Firehose 暫時保留資料，以減少最終資料儲存位置超載的機會。

工作負載

工作負載會影響叢集可以擴展的大小。使用 Kubernetes APIs工作負載會大幅限制單一叢集中可擁有的工作負載總數，但您可以變更一些預設值，以協助減少負載。

Kubernetes 叢集中的工作負載可存取與 Kubernetes API 整合的功能（例如 Secrets 和 ServiceAccounts)，但這些功能並非一律必要，如果未使用，則應停用。限制工作負載存取和對 Kubernetes 控制平面的依賴，將增加您可以在叢集中執行的工作負載數量，並透過移除不必要的工作負載存取並實作最低權限實務來提高叢集的安全性。如需詳細資訊，請參閱安全最佳實務。

使用 IPv6 進行 Pod 聯網

您無法將 VPC 從 IPv4 轉換為 IPv6，因此在佈建叢集之前啟用 IPv6 非常重要。如果您在 VPC 中啟用 IPv6，並不表示您必須使用它，而且如果您的 Pod 和服務使用 IPv6，您仍然可以路由往返 IPv4 地址的流量。如需詳細資訊，請參閱 EKS 聯網最佳實務。

在叢集中使用 IPv6 可避免一些最常見的叢集和工作負載擴展限制。IPv6 可避免因沒有可用的 IP 地址而無法建立 Pod 和節點的 IP 地址耗盡。它也改善了每個節點的效能，因為 Pod 透過減少每個節點的 ENI 附件數量，更快地接收 IP 地址。您可以在 VPC CNI 中使用 IPv4 字首模式來實現類似的節點效能，但您仍然需要確保 VPC 中有足夠的 IP 地址可用。

限制每個命名空間的服務數量

命名空間中的服務數量上限為 5，000，叢集中的服務數量上限為 10，000。為了協助組織工作負載和服務、提高效能，並避免命名空間範圍資源的層疊影響，我們建議將每個命名空間的服務數量限制為 500。

使用 kube-proxy 為每個節點建立的 IP 資料表規則數量會隨著叢集中的服務總數而增加。產生數千個 IP 資料表規則並透過這些規則路由封包會對節點產生效能影響，並新增網路延遲。

建立包含單一應用程式環境的 Kubernetes 命名空間，只要每個命名空間的服務數量低於 500。這將使服務探索夠小，以避免服務探索限制，也可以協助您避免服務命名衝突。應用程式環境（例如 dev、test、prod) 應使用單獨的 EKS 叢集，而不是命名空間。

了解 Elastic Load Balancer 配額

建立服務時，請考慮您將使用的負載平衡類型（例如 Network Load Balancer (NLB) 或 Application Load Balancer (ALB))。每個負載平衡器類型都提供不同的功能，並具有不同的配額。某些預設配額可以調整，但有一些配額上限無法變更。若要檢視您的帳戶配額和用量，請在 AWS 主控台中檢視 Service Quotas 儀表板。

例如，預設的 ALB 目標為 1000。如果您有超過 1000 個端點的服務，則需要增加配額或將服務分割到多個 ALBs 或使用 Kubernetes Ingress。預設 NLB 目標為 3000，但每個 AZ 限制為 500 個目標。如果您的叢集為 NLB 服務執行超過 500 個 Pod，您將需要使用多個 AZs 或請求提高配額限制。

使用與服務耦合的負載平衡器的替代方案是使用輸入控制器。AWS Load Balancer 控制器可以為輸入資源建立 ALBs，但您可以考慮在叢集中執行專用控制器。叢集內輸入控制器可讓您在叢集內執行反向代理，從單一負載平衡器公開多個 Kubernetes 服務。控制器具有不同的功能，例如對 Gateway API 的支援，視工作負載的數量和大小而定，可能會有好處。

使用 Route 53、Global Accelerator 或 CloudFront

若要使用多個負載平衡器做為單一端點提供服務，您需要使用 Amazon CloudFront、AWS Global Accelerator 或 Amazon Route 53，將所有負載平衡器公開為單一、面向客戶的端點。每個選項都有不同的優點，可以根據您的需求單獨或一起使用。

Route 53 可以使用通用名稱公開多個負載平衡器，並根據指派的權重將流量傳送至每個負載平衡器。您可以在文件中閱讀更多關於 DNS 權重的資訊，也可以在 AWS Load Balancer 控制器文件中閱讀如何使用 Kubernetes 外部 DNS 控制器實作它們。

Global Accelerator 可以根據請求 IP 地址，將工作負載路由到最近的區域。這對於部署到多個區域的工作負載可能很有用，但無法改善單一區域中單一叢集的路由。如果無法使用 AZ，則使用 Route 53 搭配 Global Accelerator 具有其他優點，例如運作狀態檢查和自動容錯移轉。您可以在此部落格文章中查看將 Global Accelerator 與 Route 53 搭配使用的範例。

CloudFront 可與 Route 53 和 Global Accelerator 搭配使用，也可以單獨使用，將流量路由到多個目的地。CloudFront 會快取從原始來源提供的資產，這可能會減少頻寬需求，視您服務的內容而定。

使用 EndpointSlices 而非 Endpoints

探索符合服務標籤的 Pod 時，您應該使用 EndpointSlices 而非 Endpoints。端點是公開小規模服務的一種簡單方法，但自動擴展或進行更新的大型服務會導致 Kubernetes 控制平面上的大量流量。EndpointSlices 具有自動分組功能，可啟用拓撲感知提示等物件。

根據預設，並非所有控制器都會使用 EndpointSlices。您應該驗證控制器設定，並視需要啟用它。對於 AWS Load Balancer 控制器，您應該啟用`--enable-endpoint-slices`選用旗標以使用 EndpointSlices。

如果可能，請使用不可變和外部秘密

kubelet 會針對用於節點上 Pod 磁碟區中的秘密，保留目前金鑰和值的快取。kubelet 會在 Secrets 上設定監看，以偵測變更。隨著叢集擴展，越來越多的監看可能會對 API 伺服器效能產生負面影響。

有兩種策略可以減少 Secrets 上的手錶數量：

- 對於不需要存取 Kubernetes 資源的應用程式，您可以透過設定 automountServiceAccountToken 來停用自動掛載服務帳戶秘密：false
- 如果您應用程式的秘密是靜態的，且未來不會修改，請將秘密標記為不可變。kubelet 不會維護不可變秘密的 API 監視。

若要停用自動將服務帳戶掛載到 Pod，您可以在工作負載中使用下列設定。如果特定工作負載需要服務帳戶，您可以覆寫這些設定。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app
automountServiceAccountToken: true
```

在超過 10，000 個限制之前，監控叢集中的秘密數量。您可以使用下列命令查看叢集中的秘密總數。您應該透過叢集監控工具來監控此限制。

```
kubectl get secrets -A | wc -l
```

您應該設定監控，在達到此限制之前提醒叢集管理員。考慮使用外部秘密管理選項，例如 AWS Key Management Service (AWS KMS) 或 Hashicorp Vault 搭配 Secrets Store CSI 驅動程式。

限制部署歷史記錄

建立、更新或刪除 Pod 時可能會很慢，因為叢集中仍會追蹤舊物件。您可以減少部署`revisionHistoryLimit`的 以清除較舊的 ReplicaSets，這會降低到 Kubernetes Controller Manager 追蹤的物件總數。部署的預設歷史記錄限制為 10。

如果您的叢集透過 CronJobs 或其他機制建立許多任務物件，您應該使用`ttlSecondsAfterFinished`設定自動清除叢集中的舊 Pod。這會在指定的時間長度後，從任務歷史記錄中移除成功執行的任務。

預設停用 enableServiceLinks

當 Pod 在節點上執行時，kubelet 會為每個作用中的服務新增一組環境變數。Linux 程序具有其環境的大小上限，如果您的命名空間中有太多服務，則可以達到此上限。每個命名空間的服務數量不應超過 5，000 個。之後，服務環境變數的數量會超出殼層限制，導致 Pod 在啟動時當機。

還有其他原因會導致 Pod 不應使用服務環境變數進行服務探索。環境變數名稱衝突、洩漏的服務名稱和總環境大小是幾個。您應該使用 CoreDNS 來探索服務端點。

限制每個資源的動態許可 Webhook

動態許可 Webhook 包括許可 Webhook 和變動 Webhook。它們是不屬於 Kubernetes 控制平面的 API 端點，當資源傳送到 Kubernetes API 時，會依序呼叫該端點。每個 Webhook 預設逾時為 10 秒，如果您有多個 Webhook 或任何一個逾時，可能會增加 API 請求所需的時間。

確保您的 Webhook 高度可用，特別是在 AZ 事件期間，且 failurePolicy 已正確設定為拒絕資源或忽略失敗。不需要時，請勿允許 --dry-run kubectl 命令繞過 Webhook 來呼叫 Webhook。

```
apiVersion: admission.k8s.io/v1
kind: AdmissionReview
request:
  dryRun: False
```

變更 Webhook 可以頻繁的連續修改資源。如果您有 5 個變動的 Webhook 並部署 50 個資源等，將存放每個資源的所有版本，直到壓縮執行為止，每 5 分鐘一次，以移除舊版本的修改後資源。在此案例中，當 etcd 移除取代的資源時，將會從 etcd 移除 200 個資源版本，並且根據資源的大小，可能會在 etcd 主機上使用相當大的空間，直到每 15 分鐘執行重組為止。

此重組可能會導致在 等化中暫停，這可能會對 Kubernetes API 和控制器造成其他影響。您應該避免頻繁修改大型資源，或快速連續修改數百個資源。

比較跨多個叢集的工作負載

如果您有兩個叢集應該有類似的效能，但沒有，請嘗試比較指標來識別原因。

例如，比較叢集延遲是常見的問題。這通常是由 API 請求量的差異所造成。您可以執行下列 CloudWatch LogInsight 查詢以了解差異。

```
filter @logStream like "kube-apiserver-audit"
| stats count(*) as cnt by objectRef.apiGroup, objectRef.apiVersion,
 objectRef.resource, userAgent, verb, responseStatus.code
| sort cnt desc
| limit 1000
```

您可以新增其他篩選條件來縮小範圍，例如專注於來自 的所有清單請求foo。

```
filter @logStream like "kube-apiserver-audit"
| filter verb = "list"
```

```
| filter user.username like "foo"
| stats count(*) as cnt by objectRef.apiGroup, objectRef.apiVersion,
 objectRef.resource, responseStatus.code
| sort cnt desc
| limit 1000
```

Kubernetes 擴展理論

節點與流失率

當我們討論 Kubernetes 的可擴展性時，通常會根據單一叢集中有多少節點來執行此操作。有趣的是，這很少是了解可擴展性最有用的指標。例如，具有大量但固定數量 Pod 的 5，000 個節點叢集在初始設定後不會對控制平面造成很大的壓力。不過，如果我們採用 1，000 個節點叢集，並在不到一分鐘內嘗試建立 10，000 個簡短的即時任務，它會對控制平面造成很大的持續壓力。

只需使用節點數量來了解擴展可能會誤導。最好考慮特定期間內發生的變化率（我們在此討論中使用 5 分鐘的間隔，因為這是 Prometheus 查詢預設通常會使用的間隔）。讓我們來探索為什麼根據變革速率來建構問題框架，可以讓我們更清楚要調整哪些內容以實現所需的規模。

每秒查詢數的思考

Kubernetes 為每個元件提供許多保護機制，包括 Kubelet、Scheduler、Kube Controller Manager 和 API 伺服器，以防止 Kubernetes 鏈中的下一個連結不堪負荷。例如，Kubelet 有一個標記，以特定速率調節對 API 伺服器的呼叫。這些保護機制通常是，但不總是以每秒允許的查詢或 QPS 來表示。

變更這些 QPS 設定時必須特別小心。移除一個瓶頸，例如 Kubelet 上的每秒查詢，將影響其他下游元件。這可能會且將使系統超過特定速率，因此了解和監控服務鏈的每個部分是成功擴展 Kubernetes 工作負載的關鍵。

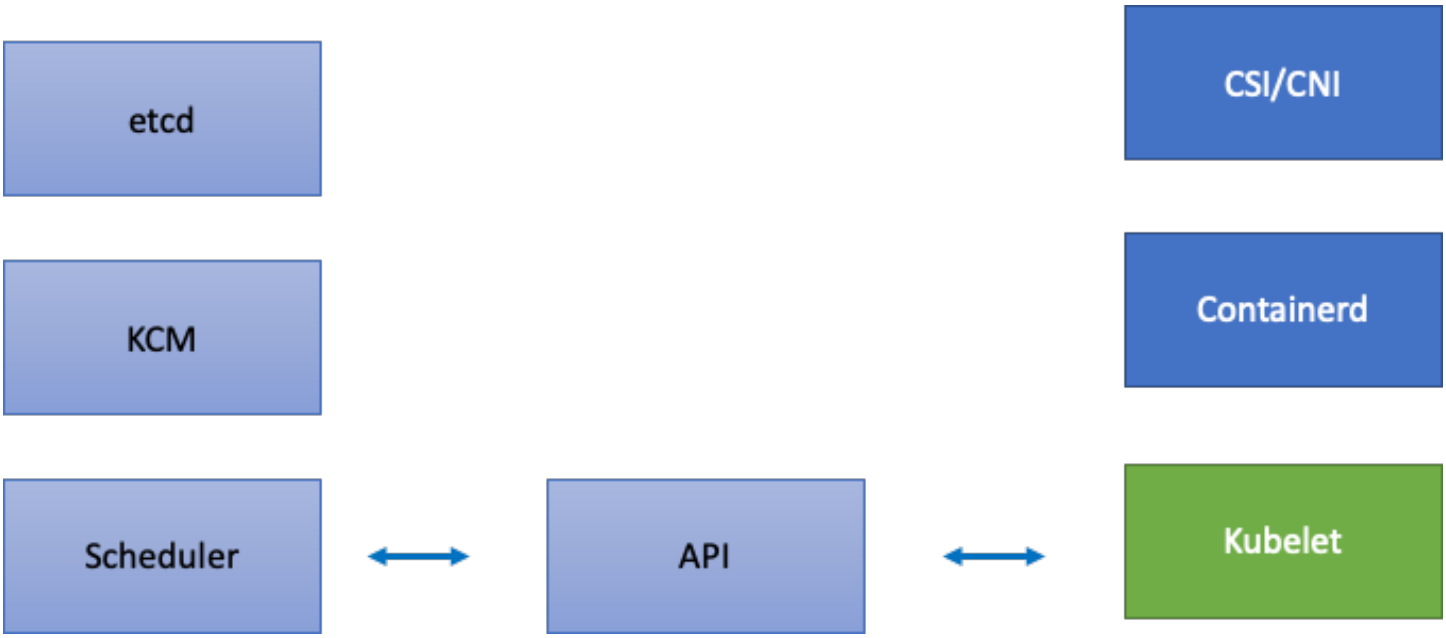
> **Note**
>
> API 伺服器具有更複雜的系統，引進 API Priority and Fairness，我們將另行討論。

> **Note**
>
> 注意，某些指標看起來很適合，但實際上是測量其他指標。例如， 只與 Kubelet 中的指標伺服器`kubelet_http_inflight_requests`相關，而不是 Kubelet 對 apiserver 請求的請求數

量。這可能會導致我們在 Kubelet 上設定錯誤 QPS 旗標。查詢特定 Kubelet 的稽核日誌會是檢查指標的更可靠方式。

擴展分散式元件

由於 EKS 是一項受管服務，讓我們將 Kubernetes 元件分為兩個類別：AWS 受管元件，包括 etcd、Kube Controller Manager 和 Scheduler（在圖表左側），以及客戶可設定的元件，例如 Kubelet、Container Runtime，以及呼叫 AWS APIs 的各種運算子，例如網路和儲存驅動程式（在圖表右側）。即使 API 伺服器受 AWS 管理，仍會將其保留在中間，因為客戶可以設定 API Priority 和 Fairness 的設定。



上游和下游瓶頸

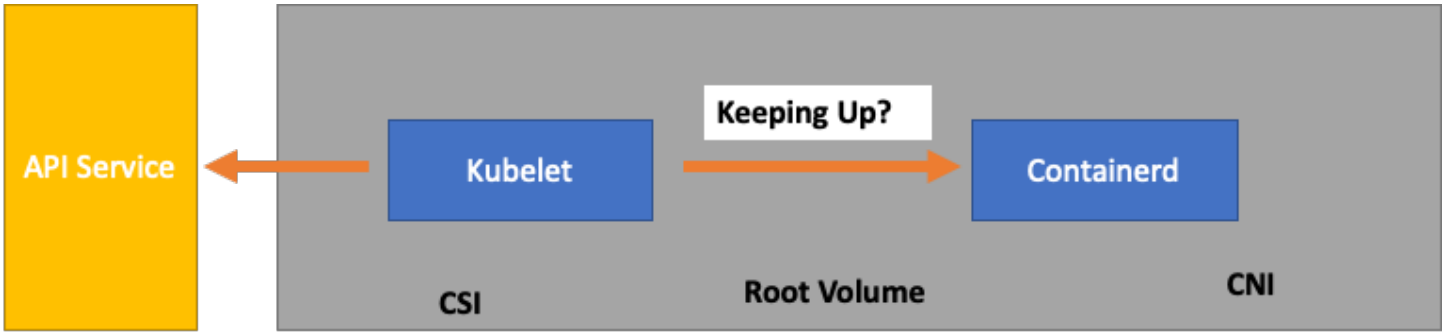
當我們監控每個服務時，請務必查看兩個方向的指標，以尋找瓶頸。讓我們了解如何使用 Kubelet 作為範例來執行此操作。Kubelet 會與 API 伺服器和容器執行期進行通訊；我們需要監控什麼和方式來偵測任一元件是否發生問題？

每個節點的 Pod 數量

當我們查看擴展數字時，例如節點上可執行的 Pod 數量時，我們可以採用上游支援的每個節點 110 個 Pod 的臉部值。

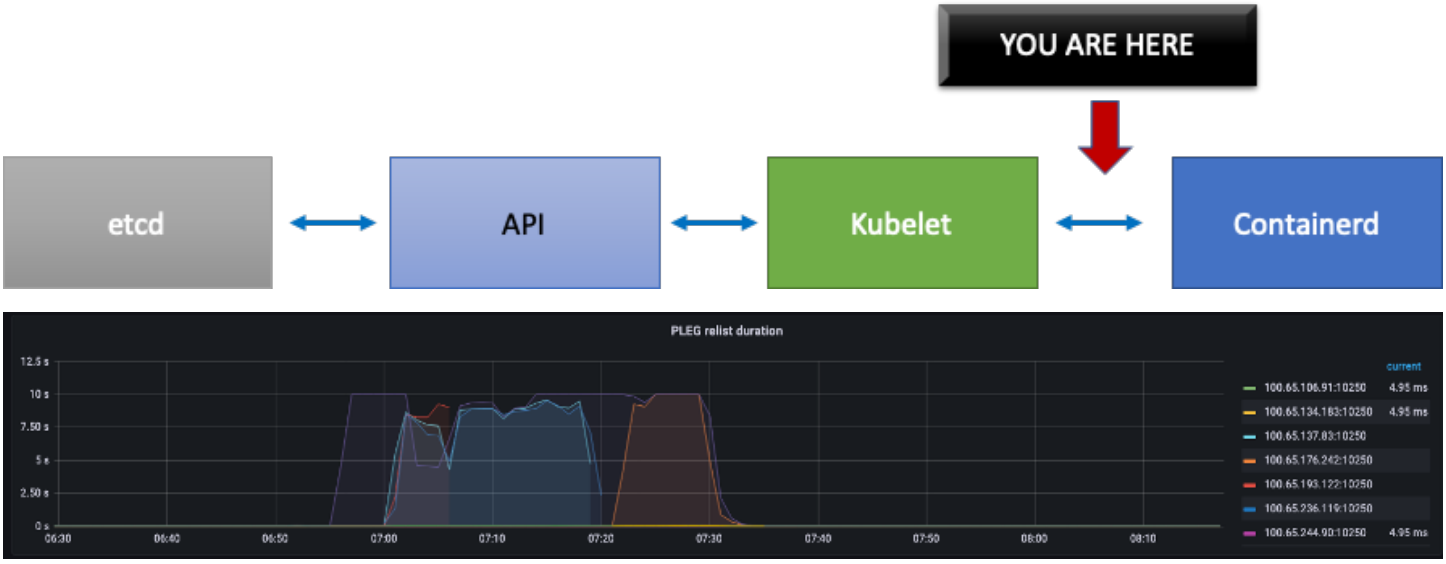
> **Note**
>
> https://kubernetes.io/docs/setup/best-practices/cluster-large/

不過，您的工作負載可能比在上游的可擴展性測試中測試的工作負載更複雜。為了確保我們可以為要在生產環境中執行的 Pod 數量提供服務，讓我們確保 Kubelet 與 Containerd 執行期保持同步。



為了過度簡化，Kubelet 正在從容器執行時間（在我們的案例中為 Containerd) 取得 Pod 的狀態。如果我們有太多 Pod 太快變更狀態，該怎麼辦？ 如果變更率太高，【對容器執行時間】 的請求可能會逾時。

> **Note**
>
> Kubernetes 不斷發展，此子系統目前正在進行變更。https：//https://github.com/kubernetes/enhancements/issues/3386



在上圖中，我們看到一條扁平線，指出我們剛達到 Pod 生命週期事件產生持續時間指標的逾時值。如果您想要在自己的叢集中看到此項目，您可以使用下列 PromQL 語法。

```
increase(kubelet_pleg_relist_duration_seconds_bucket{instance="$instance"}
[$__rate_interval])
```

如果我們目擊此逾時行為，我們知道我們已將節點推送到其能夠達到的限制。我們需要修正逾時的原因，才能繼續。這可以透過減少每個節點的 Pod 數量，或尋找可能導致大量重試（因此影響流失率）的錯誤來實現。重要的重點是，指標是了解節點是否能夠處理指派之 Pod 的流失率與使用固定數量的最佳方式。

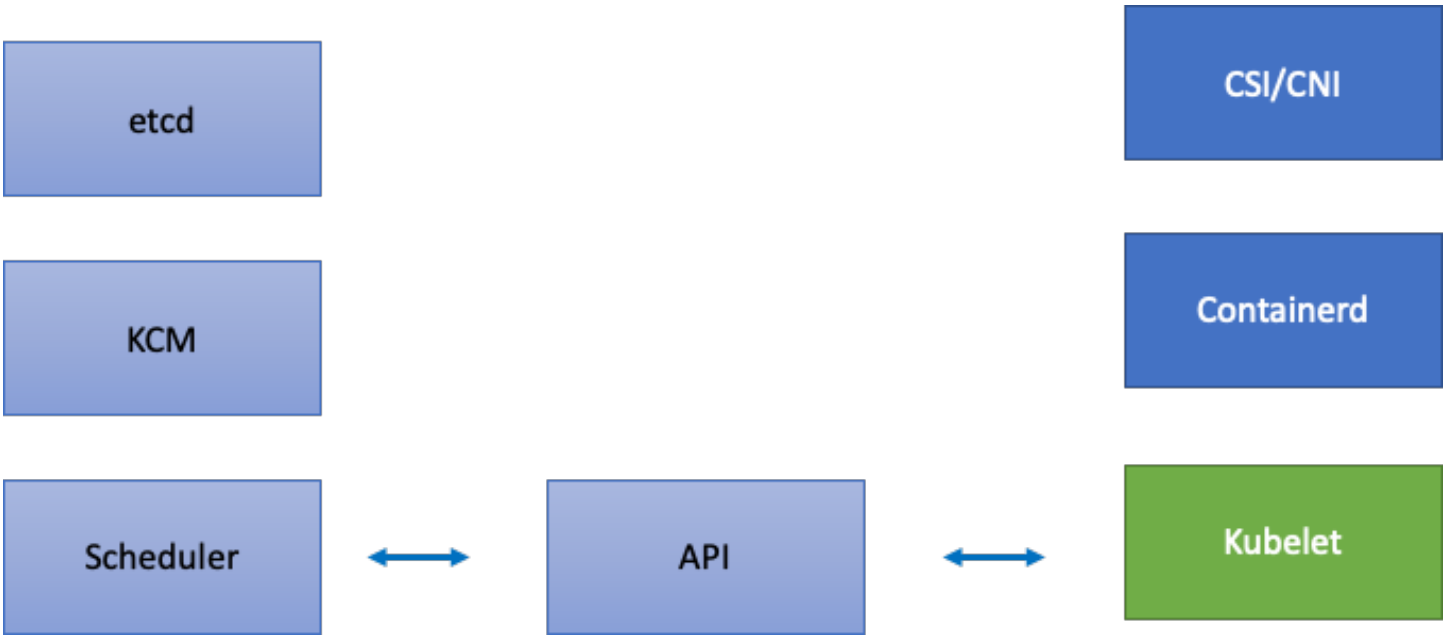
依指標擴展

雖然使用指標來最佳化系統的概念是舊的，但當人們開始 Kubernetes 旅程時，通常會被忽略。我們不是專注於特定數量（即每個節點 110 個 Pod），而是專注於尋找指標，以協助我們找出系統中的瓶頸。了解這些指標的正確閾值可以讓我們對系統進行最佳設定的高度可信度。

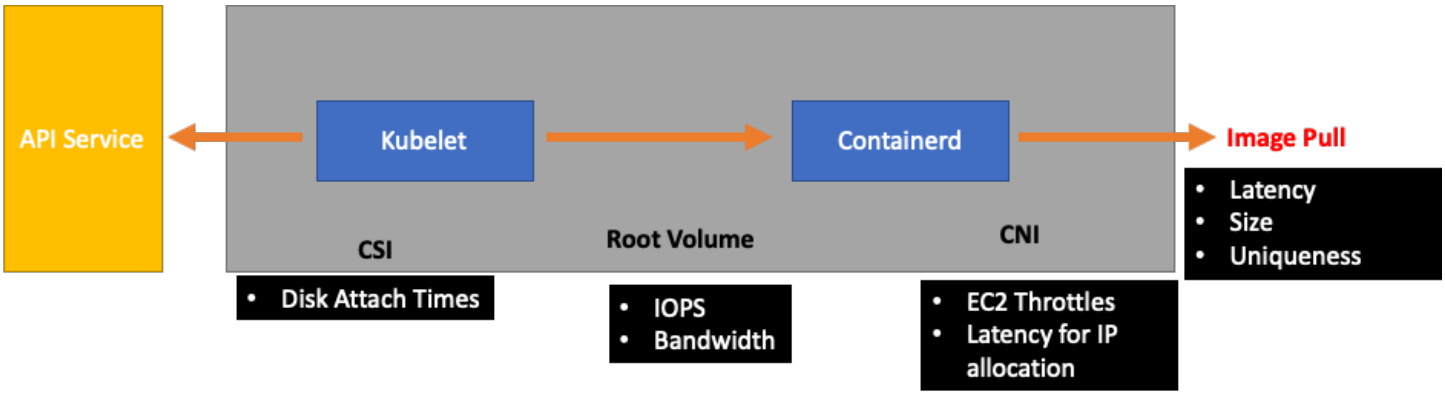
變更的影響

可能讓我們陷入麻煩的常見模式是專注於第一個看起來可疑的指標或日誌錯誤。當我們看到 Kubelet 提早逾時時，可以嘗試隨機物件，例如提高允許 Kubelet 傳送的每秒速率等。不過，最好先查看錯誤下游所有項目的全貌。以目的進行每項變更，並以資料為依據。

Kubelet 下游會是 Containerd 執行時間 (Pod 錯誤）、DaemonSets，例如儲存驅動程式 (CSI) 和與 EC2 API 通訊的網路驅動程式 (CNI) 等。



讓我們繼續先前的 Kubelet 範例，以免跟上執行時間。我們有許多點可以封裝節點，因此會密集地觸發錯誤。



為工作負載設計正確的節點大小時，這些是easy-to-overlook的訊號，可能會對系統造成不必要的壓力，進而限制我們的規模和效能。

不必要的錯誤成本

Kubernetes 控制器在發生錯誤時擅長重試，但這需要付費。這些重試可能會增加 Kube Controller Manager 等元件的壓力。這是規模測試的重要租用戶，可監控此類錯誤。

當發生較少錯誤時，系統內的 Spot 問題會比較容易。透過定期確保叢集在主要操作（例如升級）之前沒有錯誤，我們可以在發生意外事件時簡化日誌故障診斷。

擴展我們的檢視

在具有 1，000 個節點的大規模叢集中，我們不想個別尋找瓶頸。在 PromQL 中，我們可以使用稱為 topk 的函數在資料集中找到最高值；K 是我們放置所需項目數量的變數。在這裡，我們使用三個節點來了解叢集中的所有 Kubelet 是否飽和。到目前為止，我們一直在考慮延遲，現在讓我們來看看 Kubelet 是否正在捨棄事件。

```
topk(3, increase(kubelet_pleg_discard_events{}[$__rate_interval]))
```

分解此陳述式。

- 我們使用 Grafana 變數`$__rate_interval`來確保它取得所需的四個範例。這會略過使用簡單變數監控的複雜主題。
- `topk` 只給予我們最高結果，而數字 3 將這些結果限制為三個。這是適用於整個叢集指標的實用函數。

- {} 告訴我們沒有篩選條件，通常您會將任何抓取規則的任務名稱放入，但由於這些名稱不同，我們將保留空白。

將問題分割成一半

為了解決系統中的瓶頸，我們將採取方法來尋找顯示上游或下游發生問題的指標，因為這可讓我們將問題分成兩半。它也會是我們顯示指標資料的核心原則。

從此程序開始的好地方是 API 伺服器，因為它可讓我們查看用戶端應用程式或控制平面是否有問題。

控制平面監控

API 伺服器

查看我們的 API 伺服器時，請務必記住，其其中一個函數是調節傳入請求，以防止控制平面過載。API 伺服器層級的瓶頸可能實際上是保護它免受更嚴重問題的影響。我們需要考慮增加在系統中移動的請求量的優缺點。若要判斷 API 伺服器值是否應增加，以下是我們需要注意的事項的小型抽樣：

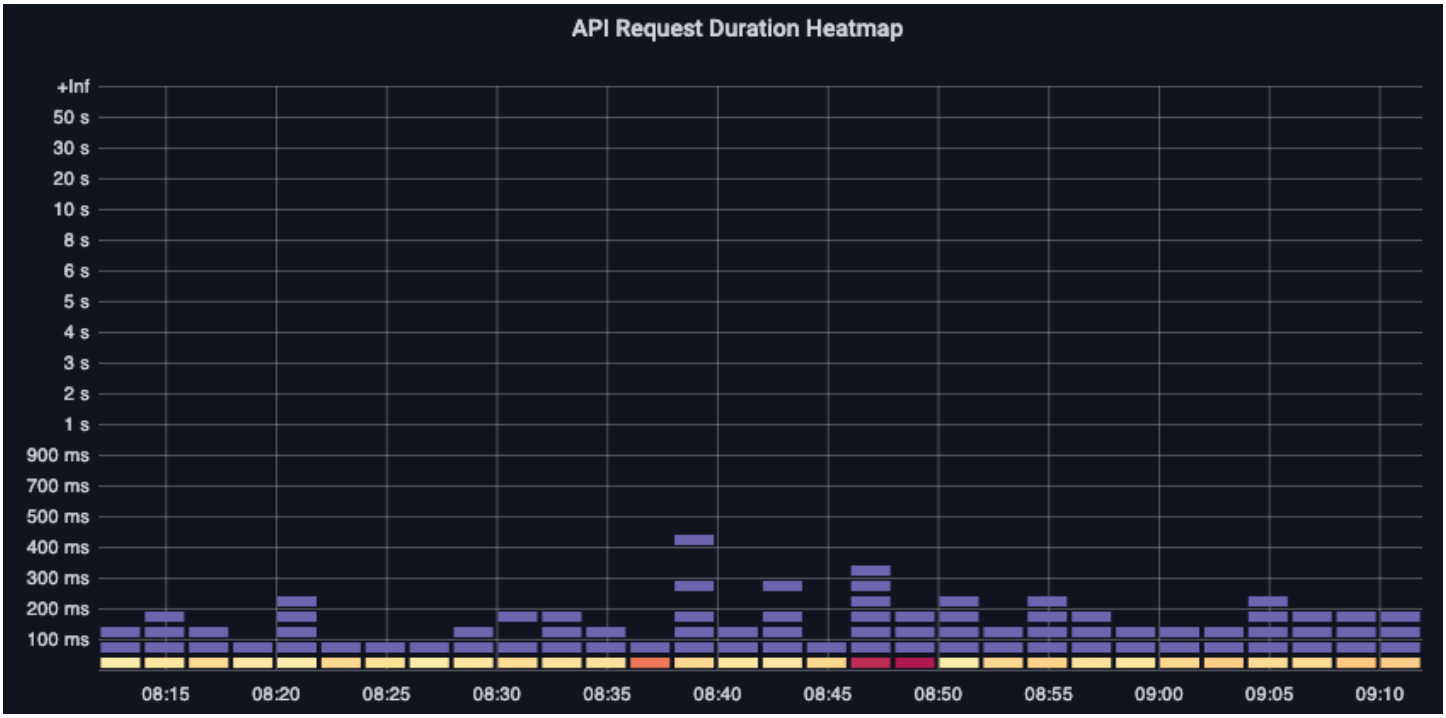
1. 請求在系統中移動的延遲為何？
2. 該延遲是 API 伺服器本身還是像是「下游」等？
3. API 伺服器佇列深度是否為此延遲的因素？
4. 針對我們想要的 API 呼叫模式，是否正確設定 API Priority and Fairness (APF) 佇列？

問題在哪裡？

首先，我們可以使用 API 延遲的 指標，讓我們深入了解 API 伺服器服務請求所需的時間。讓我們使用下列 PromQL 和 Grafana 熱度圖來顯示此資料。

```
max(increase(apiserver_request_duration_seconds_bucket{subresource!
="status",subresource!="token",subresource!="scale",subresource!="/
healthz",subresource!="binding",subresource!="proxy",verb!="WATCH"}[$__rate_interval]))
 by (le)
```

> **Note**
>
> 如需如何使用本文中使用的 API 儀表板來監控 API 伺服器的深度寫入，請參閱下列部落格



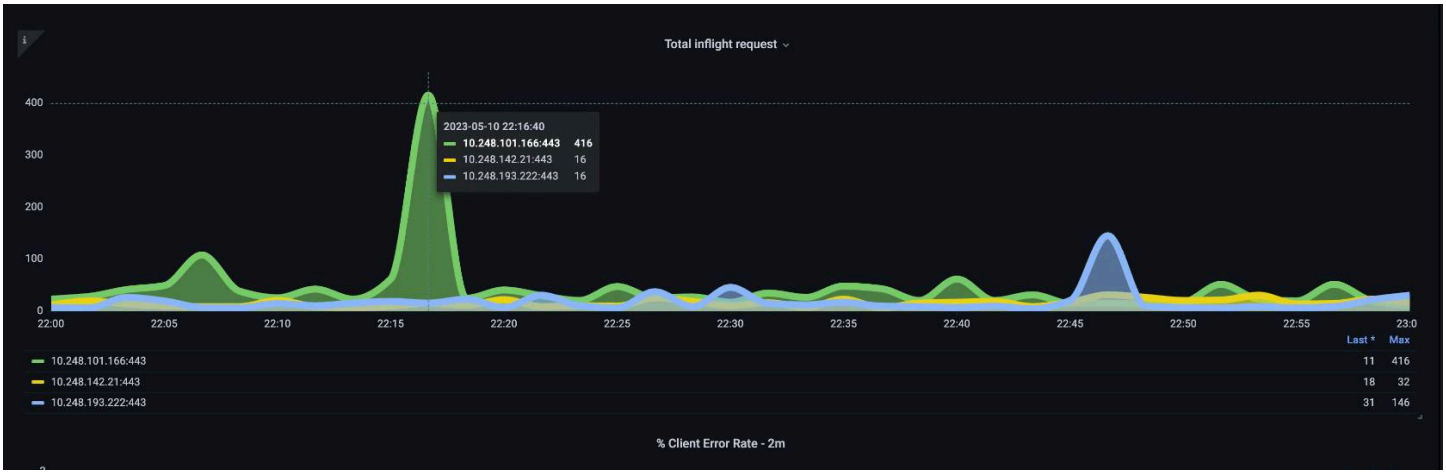
這些請求都低於一秒標記，這是控制平面及時處理請求的良好跡象。但是，如果不是這種情況，該怎麼辦？

我們在上述 API 請求持續時間中使用的格式是熱度圖。熱度圖格式的優點是，它預設會告訴我們 API 的逾時值 (60 秒）。不過，我們真正需要知道的是，在達到逾時閾值之前，此值應考量的閾值為何。如需了解哪些可接受閾值的粗略準則，我們可以使用上游 Kubernetes SLO，可在這裡找到

> **Note**
>
> 請注意此陳述式的最大函數？ 使用彙總多個伺服器的指標時（預設為 EKS 上的兩個 API 伺服器），請務必不要將這些伺服器平均在一起。

非對稱流量模式

如果一個 API 伺服器 【Pod】 被輕載，而另一個 API 伺服器被重載，該怎麼辦？ 如果我們將這兩個數字平均在一起，我們可能會誤判所發生的情況。例如，這裡有三個 API 伺服器，但所有負載都在其中一個 API 伺服器上。作為一項規則，在投資擴展和效能問題時，任何有多個伺服器，例如 etcd 和 API 伺服器，都應被分解。



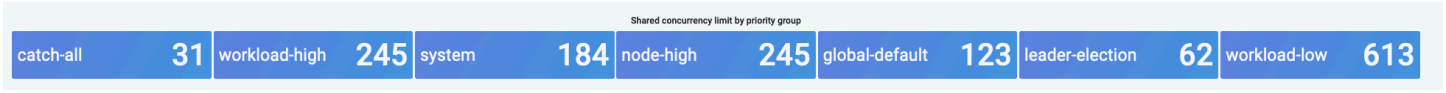
移至 API Priority and Fairness 時，系統上的請求總數只是檢查 API 伺服器是否過度訂閱的一個因素。由於系統現在可處理一系列佇列，因此我們必須查看這些佇列中是否有任何佇列已滿，以及該佇列的流量是否遭到捨棄。

讓我們使用下列查詢來查看這些佇列：

```
max without(instance)(apiserver_flowcontrol_nominal_limit_seats{})
```

> **Note**
>
> 如需 API A&F 運作方式的詳細資訊，請參閱下列最佳實務指南

在這裡，我們看到叢集上預設出現的七個不同的優先順序群組



接下來，我們想要查看使用該優先順序群組的百分比，以便了解特定優先順序層級是否飽和。調節低工作負載層級的請求可能是理想的，但不會在領導者選擇層級下降。

API Priority and Fairness (APF) 系統有許多複雜的選項，其中一些選項可能會產生意外的後果。我們在 欄位中看到的常見問題是將佇列深度增加到開始新增不必要的延遲點。我們可以使用 `apiserver_flowcontrol_current_inqueue_request` 指標來監控此問題。我們可以使用 檢查是否捨棄`apiserver_flowcontrol_rejected_requests_total`。如果任何儲存貯體超過其並行，這些指標將是非零值。

增加佇列深度可讓 API Server 成為重要的延遲來源，並應謹慎執行。我們建議您謹慎處理建立的佇列數量。例如，EKS 系統上的共用數目為 600，如果我們建立太多佇列，這可以減少需要輸送量的重要佇列中的共用，例如領導者選擇佇列或系統佇列。建立過多的額外佇列可能會讓正確調整這些佇列的大小變得更困難。

為了專注於您可以在 APF 中進行的簡單影響性變更，我們只會從未充分利用的儲存貯體取得共用，並增加使用量上限的儲存貯體大小。透過智慧地重新分配這些儲存貯體之間的共用，您可以降低捨棄的可能性。

如需詳細資訊，請參閱《EKS 最佳實務指南》中的 API 優先順序和公平性設定。

API 與 等延遲

如何使用 API 伺服器的指標/日誌來判斷 API 伺服器是否有問題，或 API 伺服器上游/下游的問題，或兩者的組合。為了更清楚這一點，讓我們看看 API Server 和 等的關聯性，以及對錯誤系統進行故障診斷的難易度。

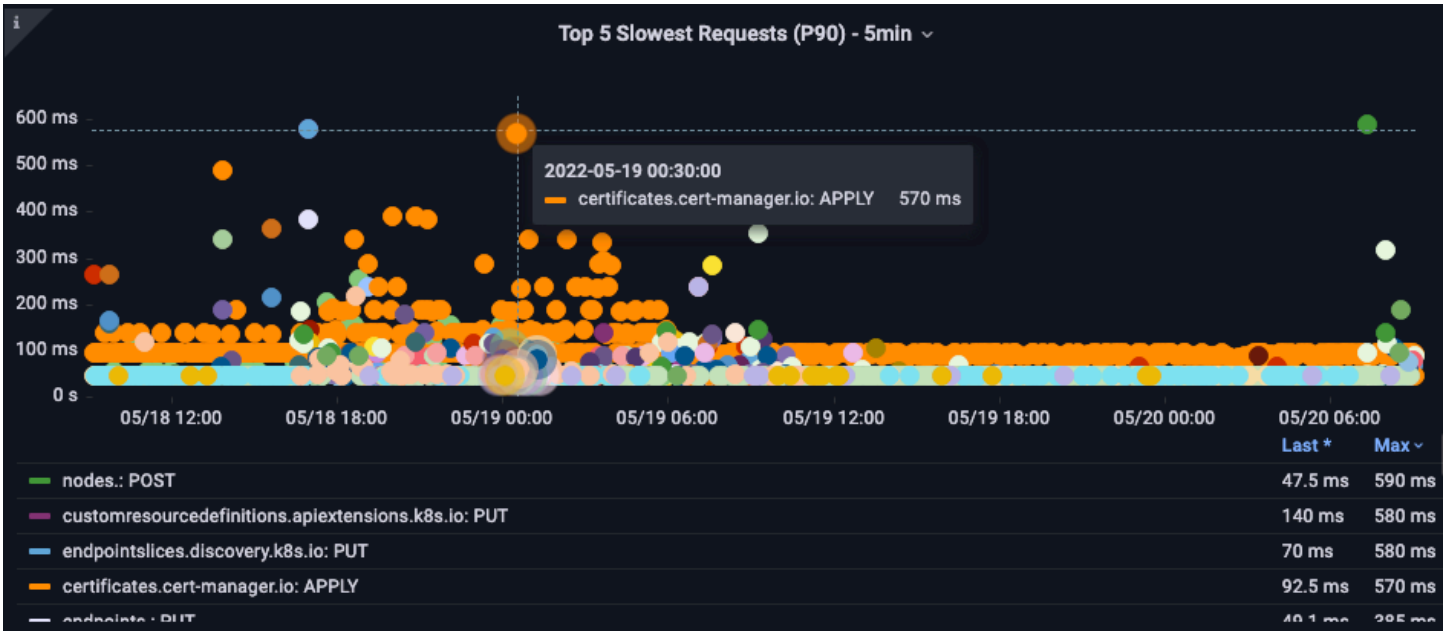
在下表中，我們看到 API 伺服器延遲，但由於圖表中的長條顯示大多數的延遲，我們也會看到大部分的延遲與等化伺服器相關。如果具有 15 秒的刻印延遲，同時有 20 秒的 API 伺服器延遲，則大多數延遲實際上是在刻印層級。

透過查看整個流程，我們發現不僅專注於 API 伺服器是明智的，但也要尋找表示被壓縮的訊號（即緩慢套用計數器增加）。只需一目了然就能快速移動到正確的問題區域，這使得儀表板功能強大。

> **Note**
>
> 您可以在 https：//https://github.com/RiskyAdventure/Troubleshooting-Dashboards/blob/main/api-troubleshooter.json 找到 區段中的儀表板

控制平面與用戶端問題

在此圖表中，我們正在尋找在該期間最花時間完成的 API 呼叫。在這種情況下，我們看到自訂資源 (CRD) 正在呼叫套用函數，這是 05：40 時間範圍內最隱含的呼叫。



透過這些資料，我們可以使用 Ad-Hoc PromQL 或 CloudWatch Insights 查詢，在該時間範圍內從稽核日誌提取 LIST 請求，以查看這可能是哪個應用程式。

使用 CloudWatch 尋找來源

指標最適合用來尋找我們想要查看並縮小問題時間範圍和搜尋參數的問題區域。取得這些資料後，我們希望轉換為日誌，以取得更詳細的時間和錯誤。為此，我們將使用 CloudWatch Logs Insights 將日誌轉換為指標。

例如，為了調查上述問題，我們將使用下列 CloudWatch Logs Insights 查詢提取 userAgent 和 requestURI，以便我們鎖定導致此延遲的應用程式。

> **Note**
>
> 必須使用適當的 Count 做為 ，才能在 Watch 上提取正常的 List/Resync 行為。

```
fields @timestamp, @message
| filter @logStream like "kube-apiserver-audit"
| filter ispresent(requestURI)
| filter verb = "list"
| parse requestReceivedTimestamp /\d+-\d+-(?<StartDay>\d+)T(?<StartHour>\d+):(?
<StartMinute>\d+):(?<StartSec>\d+).(?<StartMsec>\d+)Z/
| parse stageTimestamp /\d+-\d+-(?<EndDay>\d+)T(?<EndHour>\d+):(?<EndMinute>\d+):(?
<EndSec>\d+).(?<EndMsec>\d+)Z/
| fields (StartHour * 3600 + StartMinute * 60 + StartSec + StartMsec / 1000000) as
 StartTime, (EndHour * 3600 + EndMinute * 60 + EndSec + EndMsec / 1000000) as EndTime,
 (EndTime - StartTime) as DeltaTime
| stats avg(DeltaTime) as AverageDeltaTime, count(*) as CountTime by requestURI,
 userAgent
| filter CountTime >=50
| sort AverageDeltaTime desc
```

使用此查詢，我們找到了兩個執行大量高延遲清單操作的不同代理程式。Splunk 和 CloudWatch 代理程式。使用資料，我們可以決定移除、更新或將此控制器取代為另一個專案。

requestURI	AverageDeltaTime	AverageD	CountTime
/api/v1/pods	rest-client/2.1.0 (linux-gnu x86_64) ruby/2.5.5p157	9.1395	87
/api/v1/pods?limit=500&resourceVersion=0	amazon-cloudwatch-agent/v0.0.0 (linux/amd64) kubernetes/$Format	5.3976	929

> **Note**
>
> 如需此主題的詳細資訊，請參閱下列部落格

排程器

由於 EKS 控制平面執行個體是在不同的 AWS 帳戶中執行，因此我們將無法抓取這些元件以進行指標 (API 伺服器是例外狀況）。不過，由於我們可以存取這些元件的稽核日誌，因此可以將這些日誌轉換為指標，以查看是否有任何子系統造成擴展瓶頸。讓我們使用 CloudWatch Logs Insights 來查看排程器佇列中有多少未排程的 Pod。

排程器日誌中的未排程 Pod

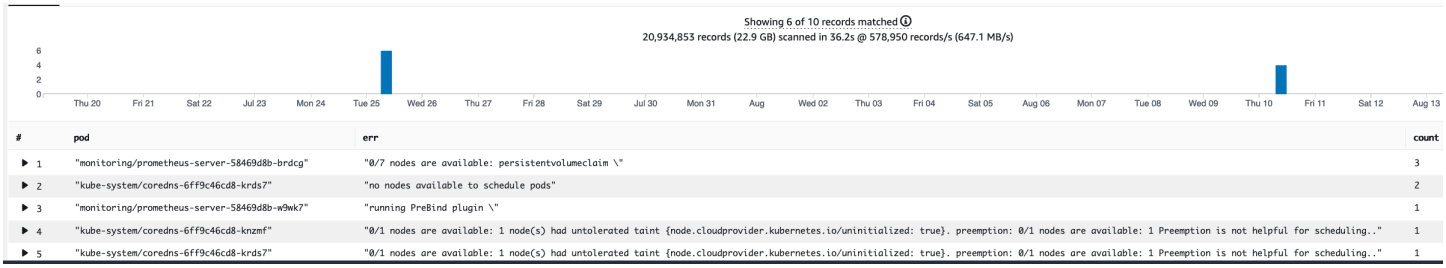
如果我們可以直接在自我管理的 Kubernetes （ 例如 Kops) 上抓取排程器指標，我們會使用下列 PromQL 來了解排程器待處理項目。

```
max without(instance)(scheduler_pending_pods)
```

由於我們無法在 EKS 中存取上述指標，因此我們會使用下列 CloudWatch Logs Insights 查詢，檢查在特定時間範圍內有多少 Pod 無法未排程，以查看待處理項目。然後，我們可以進一步深入探討尖峰時段的訊息，以了解瓶頸的性質。例如，節點旋轉速度不夠快，或排程器本身的速率限制器。

```
fields timestamp, pod, err, @message
| filter @logStream like "scheduler"
| filter @message like "Unable to schedule pod"
| parse @message  /^.(?<date>\d{4})\s+(?<timestamp>\d+:\d+:\d+\.\d+)\s+\S*\s+\S+\]\s
\"(.*?)\"\s+pod=(?<pod>\"(.*?)\")\s+err=(?<err>\"(.*?)\")/
| count(*) as count by pod, err
| sort count desc
```

在這裡，我們看到排程器的錯誤，指出 Pod 未部署，因為儲存 PVC 無法使用。



#	pod	err	count
▶ 1	"monitoring/prometheus-server-58469d8b-brdcg"	"0/7 nodes are available: persistentvolumeclaim \"	3
▶ 2	"kube-system/coredns-6ff9c46cd8-krds7"	"no nodes available to schedule pods"	2
▶ 3	"monitoring/prometheus-server-58469d8b-w9wk7"	"running PreBind plugin \"	1
▶ 4	"kube-system/coredns-6ff9c46cd8-knzmf"	"0/1 nodes are available: 1 node(s) had untolerated taint {node.cloudprovider.kubernetes.io/uninitialized: true}. preemption: 0/1 nodes are available: 1 Preemption is not helpful for scheduling.."	1
▶ 5	"kube-system/coredns-6ff9c46cd8-krds7"	"0/1 nodes are available: 1 node(s) had untolerated taint {node.cloudprovider.kubernetes.io/uninitialized: true}. preemption: 0/1 nodes are available: 1 Preemption is not helpful for scheduling.."	1

> **Note**
>
> 稽核記錄必須開啟控制平面，才能啟用此函數。這也是最佳實務，限制日誌保留，以免不必要地隨著時間推移增加成本。使用以下 EKSCTL 工具開啟所有記錄函數的範例。

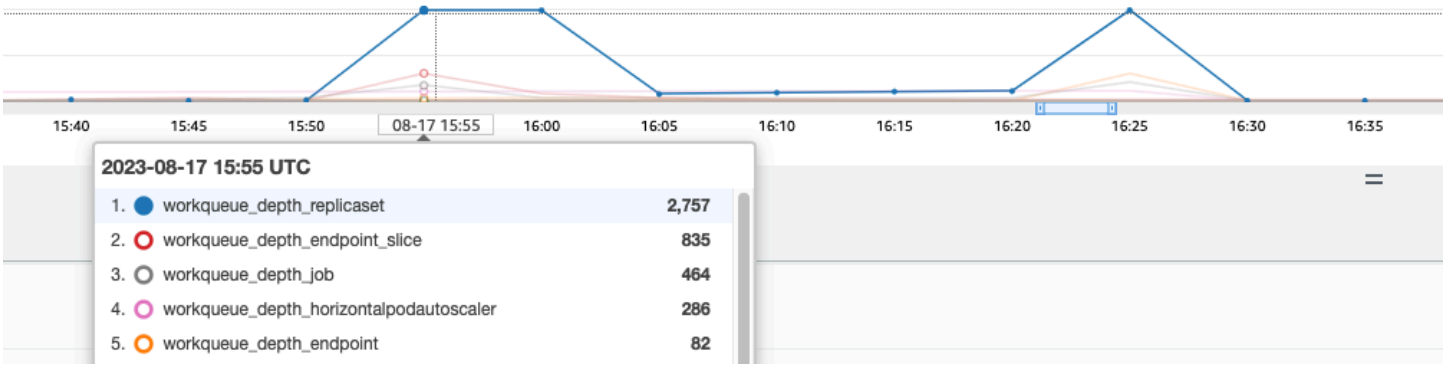
```
cloudWatch:
  clusterLogging:
    enableTypes: ["*"]
    logRetentionInDays: 10
```

Kube 控制器管理員

與所有其他控制器一樣，Kube Controller Manager 會限制一次可以執行的操作數量。讓我們看看我們可以設定這些參數的 KOPS 組態，來檢閱這些旗標有哪些。

```
kubeControllerManager:
  concurrentEndpointSyncs: 5
  concurrentReplicasetSyncs: 5
  concurrentNamespaceSyncs: 10
  concurrentServiceaccountTokenSyncs: 5
  concurrentServiceSyncs: 5
  concurrentResourceQuotaSyncs: 5
  concurrentGcSyncs: 20
  kubeAPIBurst: 30
  kubeAPIQPS: "20"
```

這些控制器具有在叢集上的高流失期間填滿的佇列。在此情況下，我們看到複本集控制器的佇列中有大型待處理項目。



我們有兩種不同的方法來解決這種情況。如果執行自我管理，我們可以簡單地增加並行 goroutine，但這會因為在 KCM 中處理更多資料而影響到 等。另一個選項是減少部署`.spec.revisionHistoryLimit`上使用 的複本集物件數量，以減少我們可以轉返的複本集物件數量，從而降低此控制器的壓力。

```
spec:
  revisionHistoryLimit: 2
```

其他 Kubernetes 功能可以調校或關閉，以減少高流失率系統中的壓力。例如，如果我們 Pod 中的應用程式不需要直接與 k8s API 交談，則關閉這些 Pod 中投影的秘密會減少 ServiceaccountTokenSyncs 的負載。如果可能，這是解決此類問題更理想的方法。

```
kind: Pod
spec:
  automountServiceAccountToken: false
```

在無法存取指標的系統中，我們可以再次查看日誌以偵測爭用。如果我們希望查看每個控制器或彙總層級上正在處理的請求數量，我們會使用下列 CloudWatch Logs Insights 查詢。

KCM 處理的磁碟區總數

```
# Query to count API qps coming from kube-controller-manager, split by controller type.
# If you're seeing values close to 20/sec for any particular controller, it's most
 likely seeing client-side API throttling.
fields @timestamp, @logStream, @message
| filter @logStream like /kube-apiserver-audit/
| filter userAgent like /kube-controller-manager/
# Exclude lease-related calls (not counted under kcm qps)
| filter requestURI not like "apis/coordination.k8s.io/v1/namespaces/kube-system/
leases/kube-controller-manager"
# Exclude API discovery calls (not counted under kcm qps)
| filter requestURI not like "?timeout=32s"
# Exclude watch calls (not counted under kcm qps)
| filter verb != "watch"
# If you want to get counts of API calls coming from a specific controller, uncomment
 the appropriate line below:
# | filter user.username like "system:serviceaccount:kube-system:job-controller"
# | filter user.username like "system:serviceaccount:kube-system:cronjob-controller"
# | filter user.username like "system:serviceaccount:kube-system:deployment-controller"
# | filter user.username like "system:serviceaccount:kube-system:replicaset-controller"
# | filter user.username like "system:serviceaccount:kube-system:horizontal-pod-
autoscaler"
# | filter user.username like "system:serviceaccount:kube-system:persistent-volume-
binder"
# | filter user.username like "system:serviceaccount:kube-system:endpointslice-
controller"
# | filter user.username like "system:serviceaccount:kube-system:endpoint-controller"
# | filter user.username like "system:serviceaccount:kube-system:generic-garbage-
controller"
| stats count(*) as count by user.username
| sort count desc
```

這裡的關鍵要點是在查看可擴展性問題時，先查看路徑中的每個步驟 (API、排程器、KCM 等），然後再移至詳細的故障診斷階段。通常在生產環境中，您會發現需要調整 Kubernetes 的多個部分，以允許系統以其效能最佳的方式運作。不小心疑難排解什麼只是較大瓶頸的症狀（例如節點逾時）很容易。

ETCD

etcd 使用記憶體映射檔案來有效率地存放金鑰值對。有一種保護機制可設定此可用記憶體空間的大小，通常設定為 2、4 和 8GB 限制。資料庫中的物件較少，表示更新物件且需要清除較舊版本時，需要執行的清除等動作較少。此清除物件舊版本的程序稱為壓縮。在多次壓縮操作之後，有一項後續程序會復原稱為重組的可用空間，該空間發生在超過特定閾值或固定的時間排程。

我們可以執行幾個與使用者相關的項目，以限制 Kubernetes 中的物件數量，從而減少壓縮和重組程序的影響。例如，Helm 會保留較高的 `revisionHistoryLimit`。這可讓系統中 ReplicaSets 等較舊的物件能夠進行轉返。透過將歷史記錄限制設定為 2，我們可以將物件數量（例如 ReplicaSets) 從 10 個減少到 2 個，進而減少對系統的負載。

```
apiVersion: apps/v1
kind: Deployment
spec:
  revisionHistoryLimit: 2
```

從監控的角度來看，如果系統延遲激增發生在以小時分隔的集合模式中，請檢查此重組程序是否有助於來源。我們可以使用 CloudWatch Logs 來查看。

如果您想要查看重組的開始/結束時間，請使用下列查詢：

```
fields @timestamp, @message
| filter @logStream like /etcd-manager/
| filter @message like /defraging|defraged/
| sort @timestamp asc
```

@message	@logStream
Oct 6 19:19:52 ip-172-16-176-9 etcdnanny: {"level":"info","ts":"2022-10-06T19:19:52.826Z","caller":"defrag/defrag.go:312","msg":"defraging","…	etcd-manager-i-0101a20d88e74fa5b
Oct 6 19:20:28 ip-172-16-176-9 etcdnanny: {"level":"info","ts":"2022-10-06T19:20:28.836Z","caller":"defrag/defrag.go:322","msg":"defraged"}	etcd-manager-i-0101a20d88e74fa5b
Oct 6 19:25:57 ip-172-16-101-191 etcdnanny: {"level":"info","ts":"2022-10-06T19:25:57.650Z","caller":"defrag/defrag.go:312","msg":"defraging"…	etcd-manager-i-0e3aa5c2ae801af44
Oct 6 19:26:34 ip-172-16-101-191 etcdnanny: {"level":"info","ts":"2022-10-06T19:26:34.457Z","caller":"defrag/defrag.go:322","msg":"defraged"}	etcd-manager-i-0e3aa5c2ae801af44
Oct 6 19:32:26 ip-172-16-50-136 etcdnanny: {"level":"info","ts":"2022-10-06T19:32:26.592Z","caller":"defrag/defrag.go:312","msg":"defraging",…	etcd-manager-i-025ebb6b463ee0f21
Oct 6 19:33:09 ip-172-16-50-136 etcdnanny: {"level":"info","ts":"2022-10-06T19:33:09.575Z","caller":"defrag/defrag.go:322","msg":"defraged"}	etcd-manager-i-025ebb6b463ee0f21

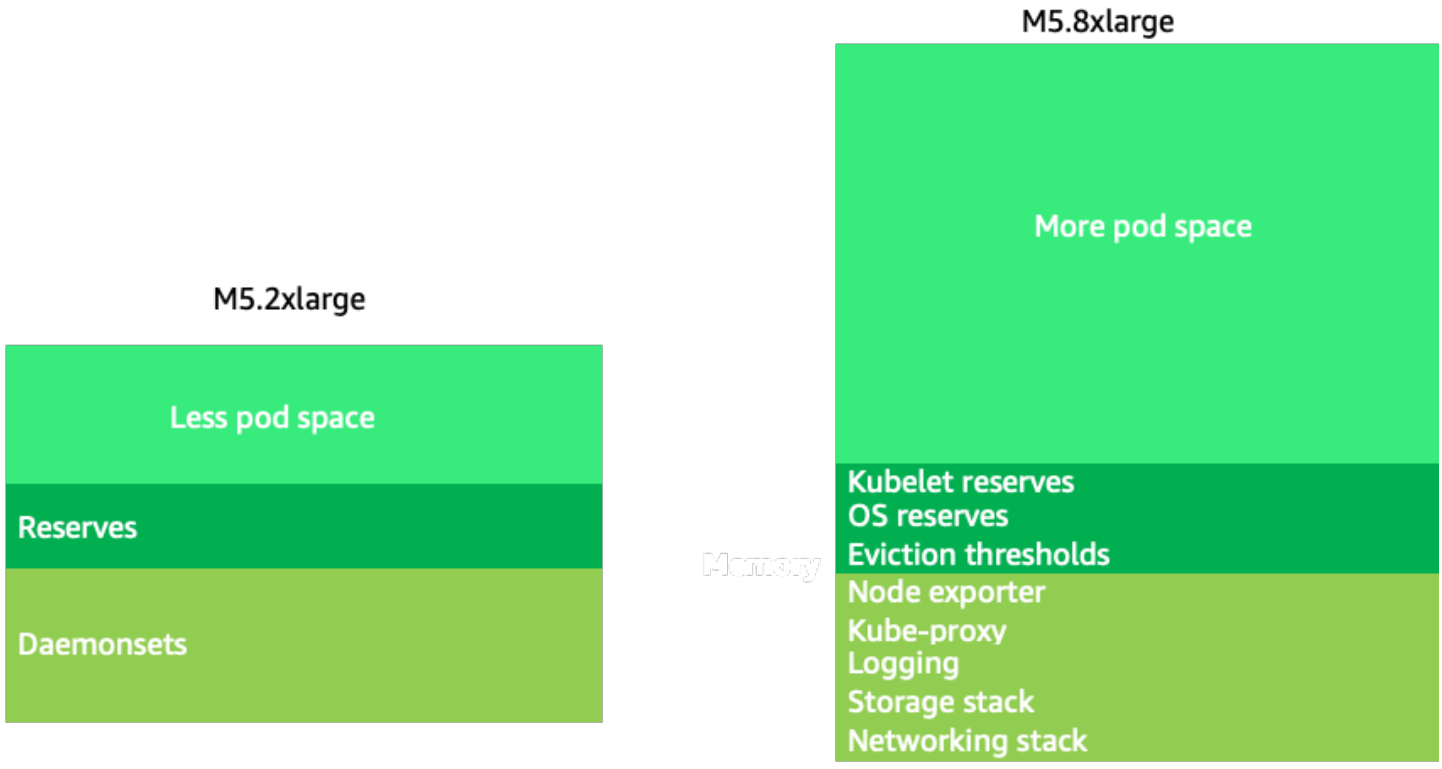
節點和工作負載效率

有效率地處理工作負載和節點可降低複雜性/成本，同時提高效能和規模。規劃此效率時需要考慮許多因素，而且最簡單的是考慮權衡與每個功能的最佳實務設定。讓我們在下一節深入探索這些權衡。

節點選擇

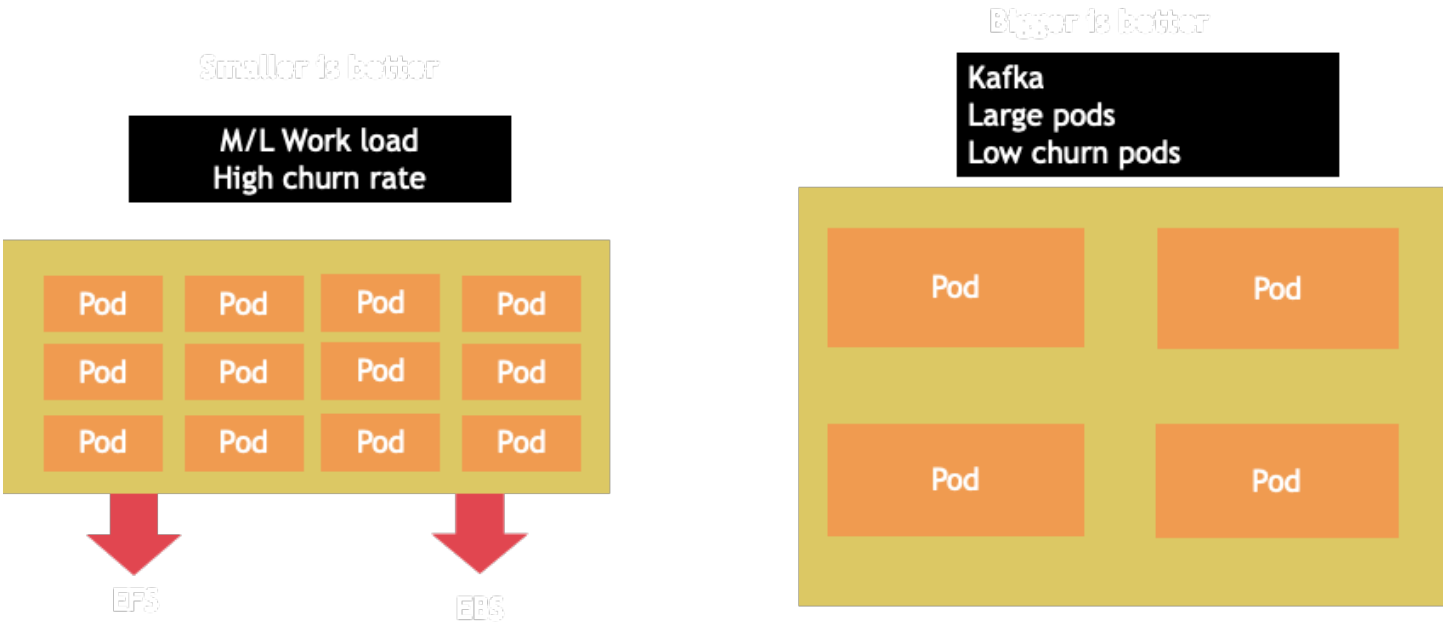
使用略大的節點大小 (4-12xlarge) 會增加我們用來執行 Pod 的可用空間，因為它可減少用於「額外負荷」的節點百分比，例如 DaemonSets 和 預留系統元件。在下圖中，我們看到 2xlarge 上的可用空間與只有中等數量的 DaemonSets 的 8xlarge 系統之間的差異。

> **Note**
>
> 由於 k8s 會水平擴展為一般規則，因此對於大多數應用程式而言，採取 NUMA 大小節點的效能影響並不合理，因此建議低於該節點大小的範圍。



大型節點大小可讓我們在每個節點擁有較高百分比的可用空間。不過，此模型可以透過使用太多 Pod 封裝節點來導致錯誤或使節點飽和，進而達到極端。監控節點飽和度是成功使用較大節點大小的關鍵。

節點選擇很少是 one-size-fits-all 提案。通常最好將具有顯著不同流失率的工作負載分割為不同的節點群組。4xlarge 系列執行個體最適合提供具有高流失率的小型批次工作負載，而 Kafka 等大型應用程式需要 8 個 vCPU，而 12xlarge 系列更適合提供低流失率。



> **Note**
>
> 另一個考慮非常大節點大小的因素是，因為 CGROUPS 不會從容器化應用程式中隱藏 vCPU 總數。動態執行時間通常會產生意外數量的作業系統執行緒，造成難以進行故障診斷的延遲。對於這些應用程式，建議使用 CPU 鎖定。如需主題的深入探索，請參閱下列影片 https：// https://www.youtube.com/watch？v=NqtfDy_KAqg

節點儲存貯體封裝

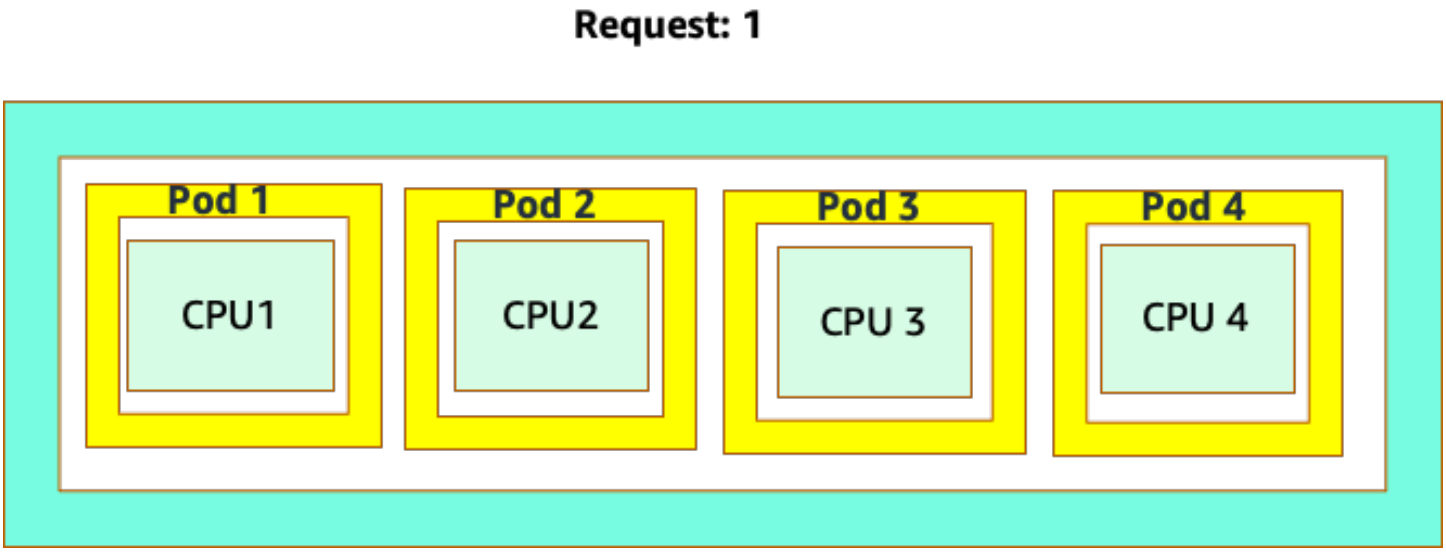
Kubernetes 與 Linux 規則

在 Kubernetes 上處理工作負載時，需要注意兩組規則。Kubernetes 排程器的規則，使用請求值來排程節點上的 Pod，然後排程 Pod 之後會發生什麼情況，這是 Linux 的領域，而不是 Kubernetes。

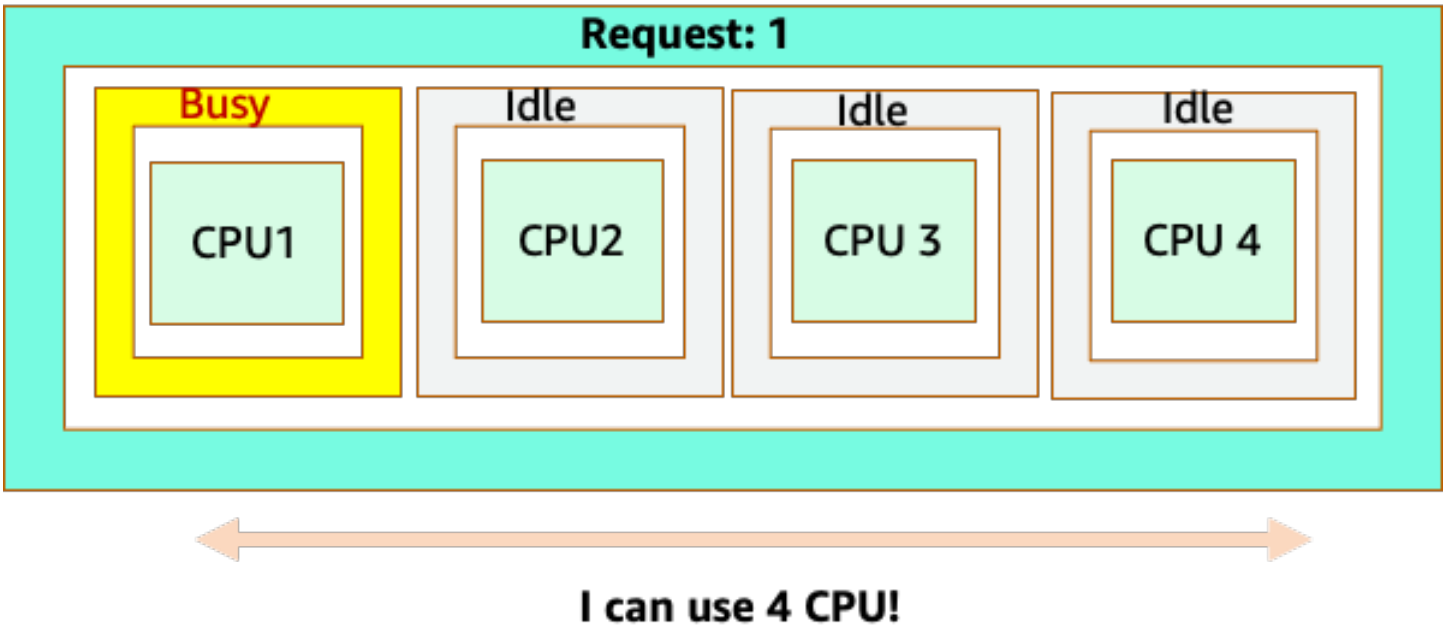
Kubernetes 排程器完成後，一組新的規則接管 Linux 完全公平排程器 (CFS)。關鍵點是 Linux CFS 沒有核心的概念。我們將討論為什麼在核心中思考可能會導致最佳化規模工作負載的重大問題。

在 核心中思考

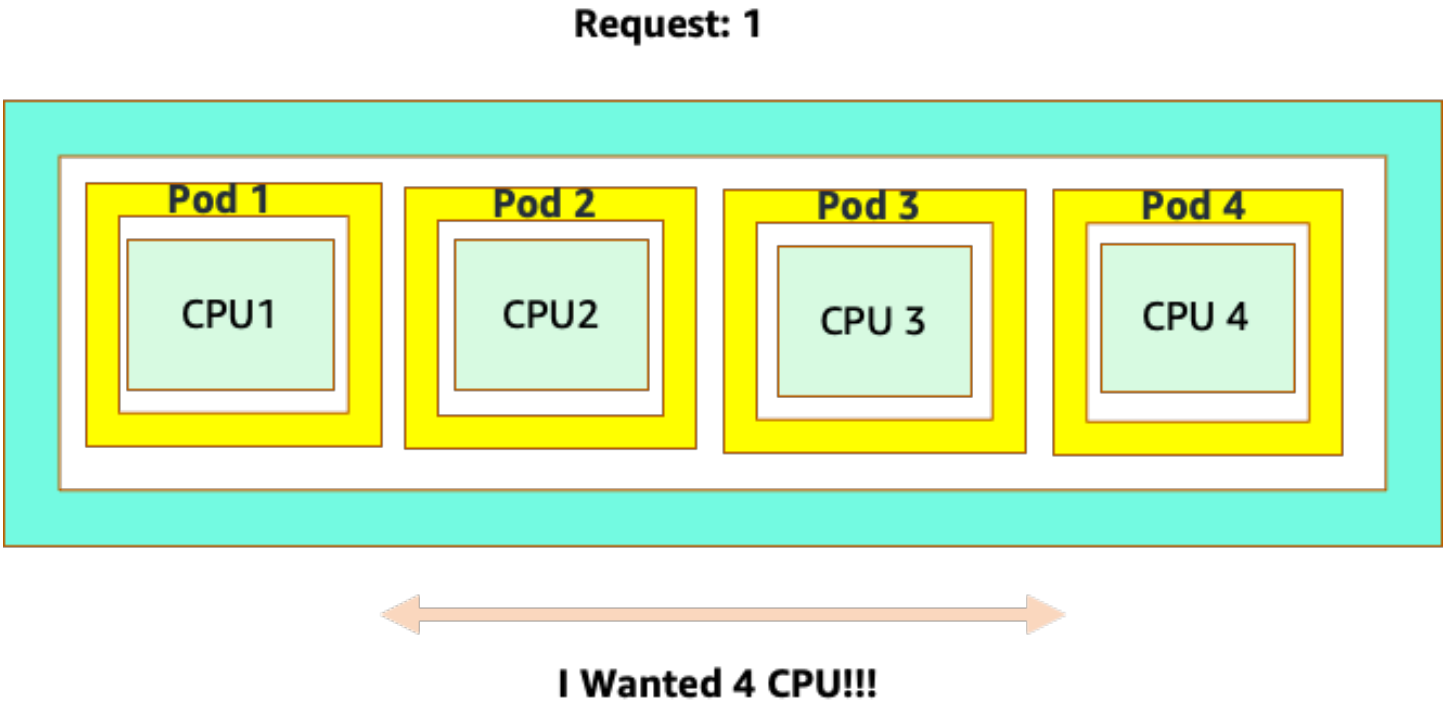
混淆開始是因為 Kubernetes 排程器確實具有核心的概念。從 Kubernetes 排程器的角度來看，如果我們查看具有 4 個 NGINX Pod 的節點，每個節點都有一個核心集的請求，節點看起來會像這樣。



不過，讓我們進行一個思維實驗，了解這與 LinuxCFS 的角度有何不同。使用 Linux MCU 系統時要記住的最重要事項是：忙碌容器 (CGROUPS) 是唯一計入共用系統的容器。在這種情況下，只有第一個容器處於忙碌狀態，因此允許它在節點上使用所有 4 個核心。



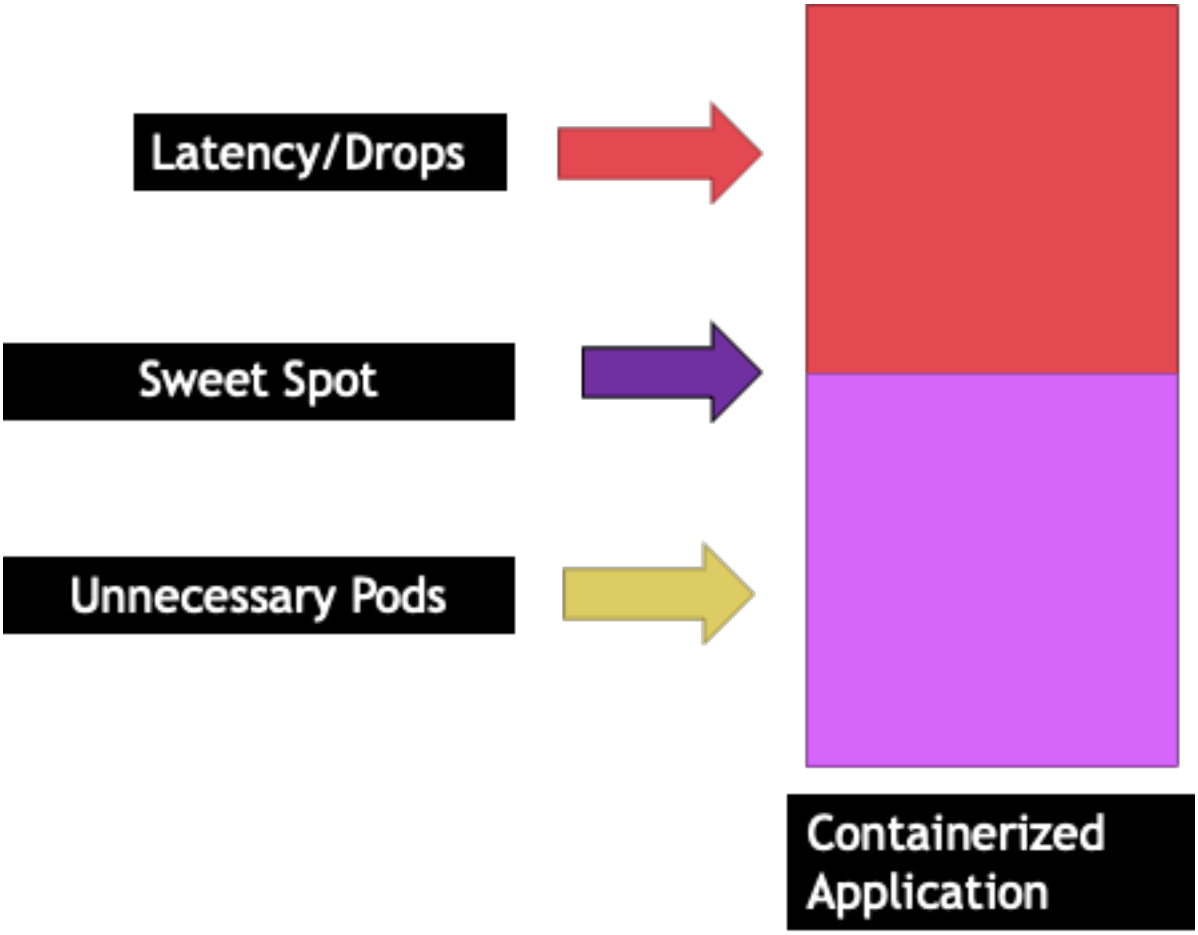
為什麼這很重要？ 假設我們在開發叢集中執行效能測試，其中 NGINX 應用程式是該節點上唯一忙碌的容器。當我們將應用程式移至生產環境時，會發生下列情況：NGINX 應用程式需要 4 個 vCPU 的資源，但由於節點上的所有其他 Pod 都忙碌中，因此我們應用程式的效能受到限制。



這種情況會導致我們不必要地新增更多容器，因為我們不允許應用程式擴展到「sweet spot」`。讓我們更詳細"sweet spot"地探索 的這個重要概念。

應用程式正確調整大小

每個應用程式都有無法再接受流量的特定點。超過此點可能會增加處理時間，甚至會在推送遠超過此點時捨棄流量。這稱為應用程式的飽和點。為了避免擴展問題，我們應該在應用程式達到飽和點之前嘗試擴展應用程式。讓我們將此點稱為甜點。

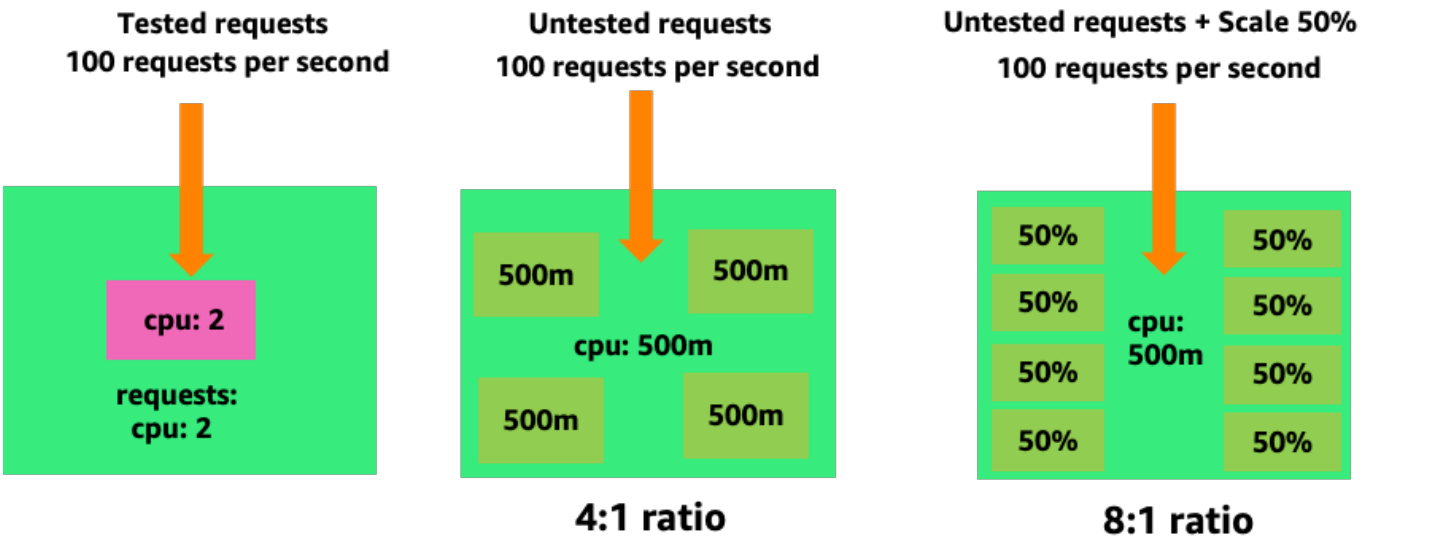


我們需要測試每個應用程式，以了解其甜點。這裡不會有通用指引，因為每個應用程式都不同。在此測試期間，我們嘗試了解顯示應用程式飽和點的最佳指標。使用率指標通常用於表示應用程式飽和，但這可能會快速導致擴展問題（我們將在稍後的章節中詳細探討此主題）。一旦我們擁有這個 "`sweet spot"`，就可以使用它來有效率地擴展工作負載。

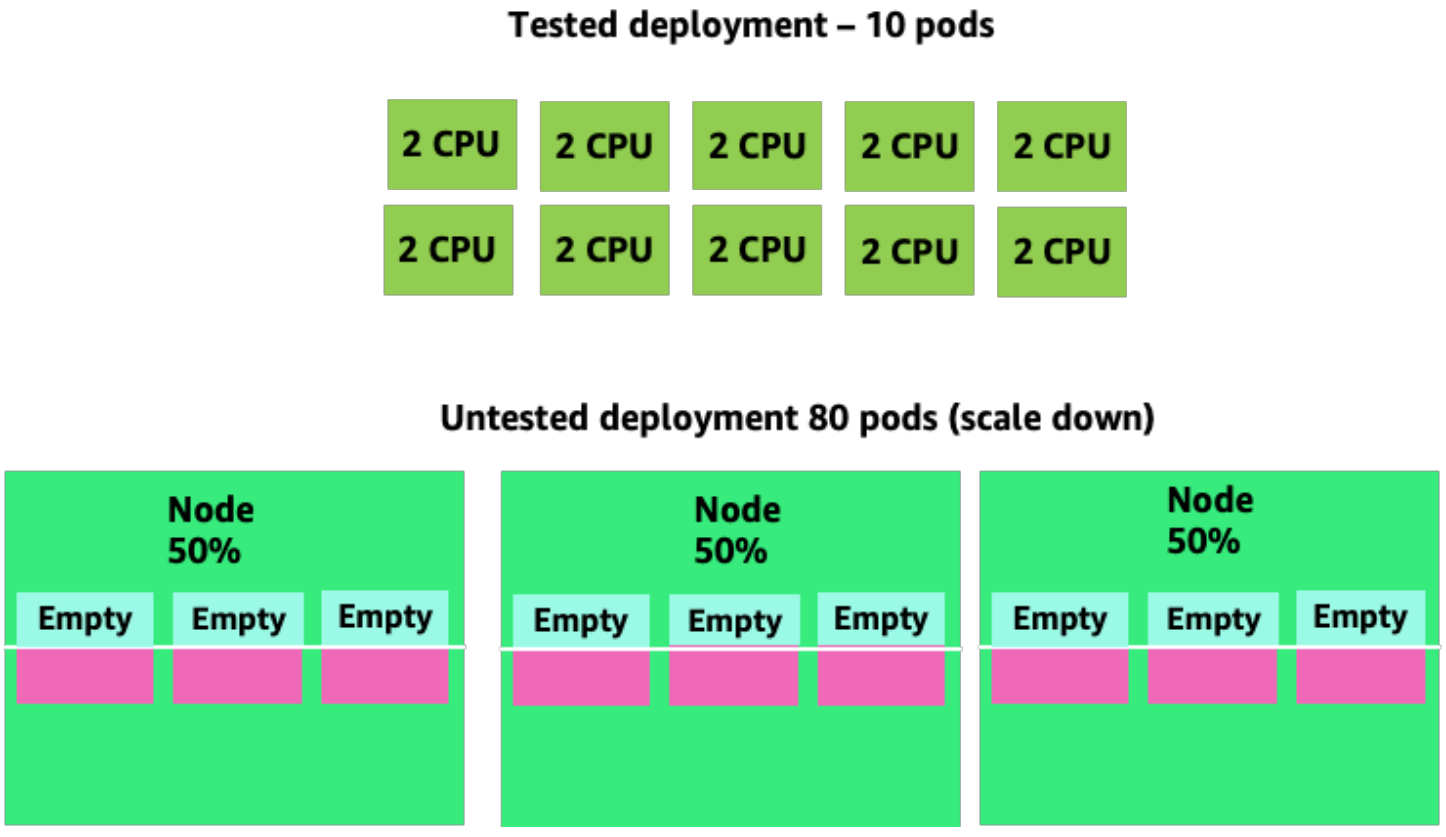
相反地，如果我們在甜點之前順利擴展並造成不必要的 Pod，會發生什麼情況？ 讓我們在下一節探索。

Pod 擴展

若要了解建立不必要的 Pod 如何快速失控，讓我們來看看左側的第一個範例。處理每秒 100 個請求時，此容器的正確垂直擴展會佔用大約兩個 vCPUs 使用率。不過，如果我們透過將請求設定為一半核心來少佈建請求值，現在我們實際需要的每個 Pod 都需要 4 個 Pod。進一步加劇此問題，如果我們的 HPA 設定為預設的 50% CPU，這些 Pod 會擴展一半的空白，建立 8：1 的比例。

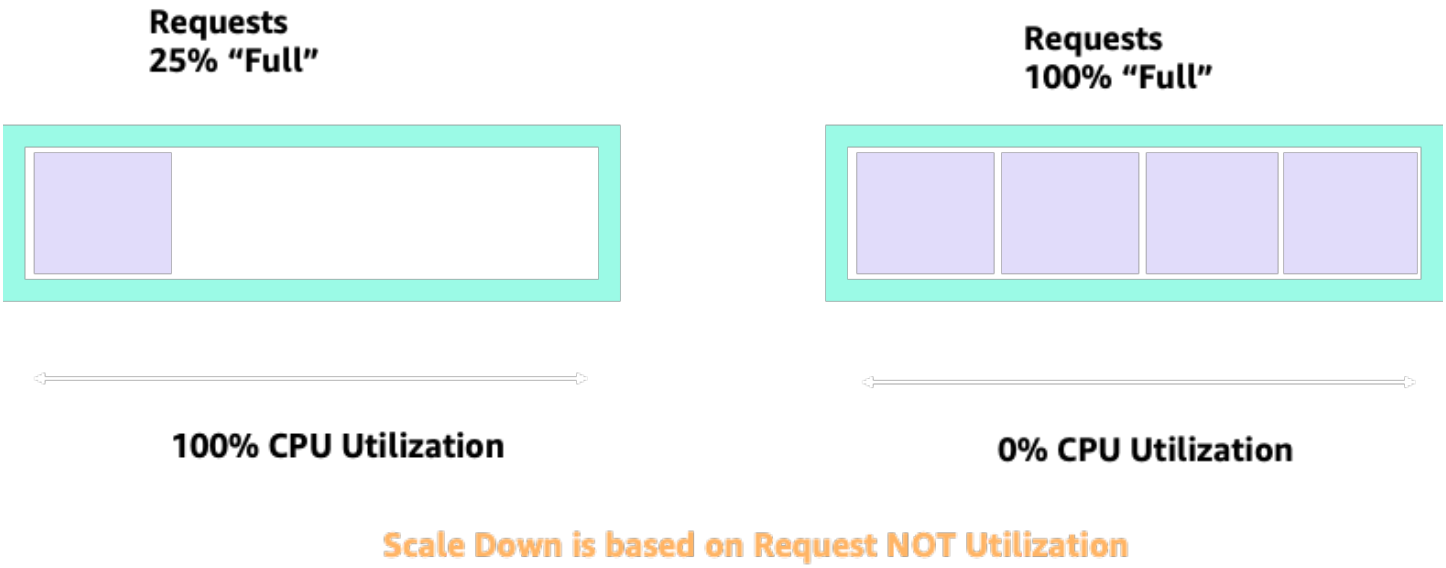


向上擴展此問題，我們可以快速了解此問題如何解決。部署 10 個 Pod，其甜點設定不正確，可能會快速旋轉至 80 個 Pod，以及執行它們所需的額外基礎設施。



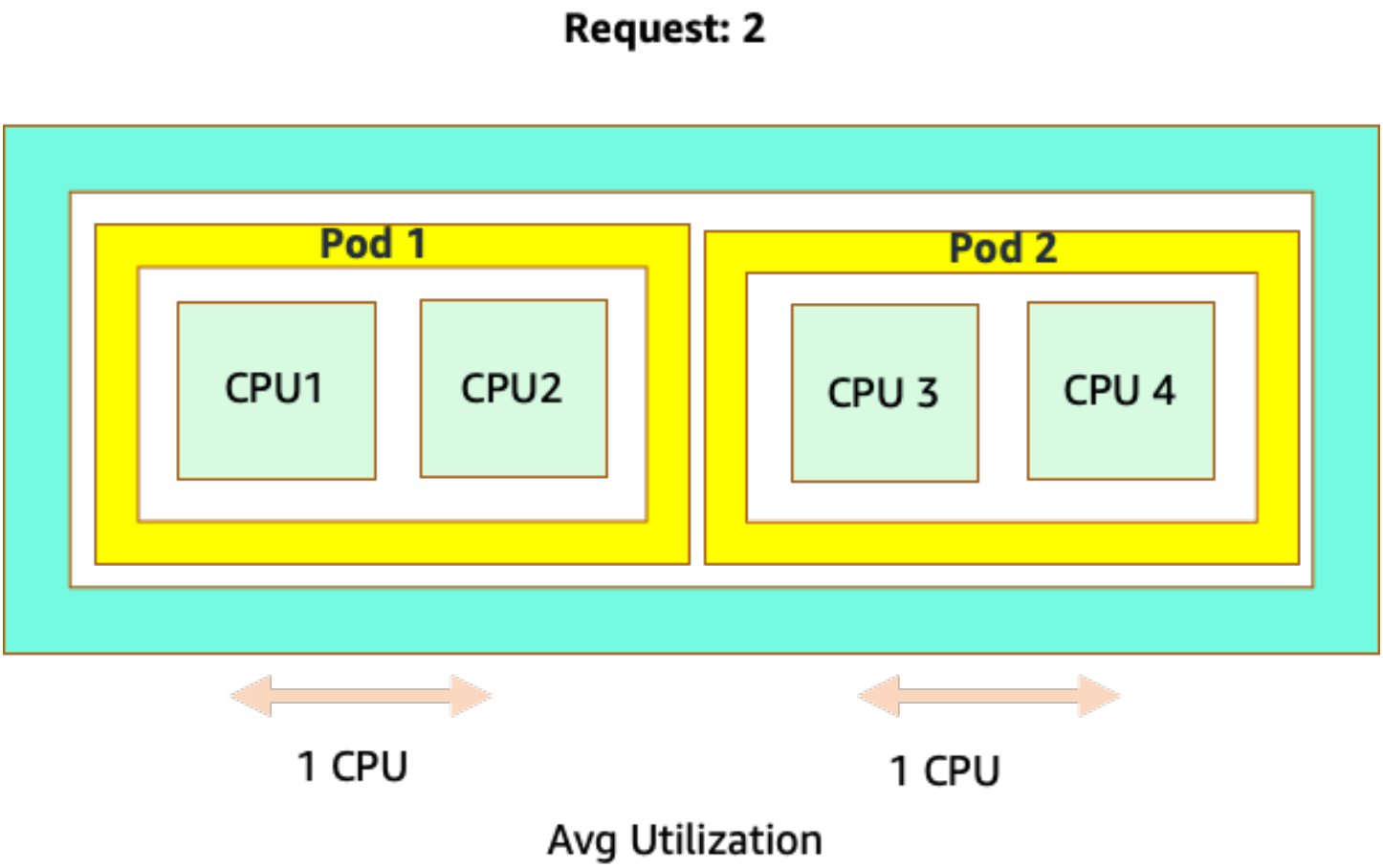
現在我們了解不允許應用程式在其甜甜圈中運作的影響，讓我們回到節點層級，並詢問 Kubernetes 排程器和 Linux MCU 之間的這種差異為什麼如此重要？

使用 HPA 擴展和縮減規模時，我們可能會有許多空間來配置更多 Pod 的案例。這是一個錯誤的決策，因為左側描述的節點已經達到 100% CPU 使用率。在不切實際但理論上可能的情況下，我們可以有另一個極端的節點完全滿，但 CPU 使用率為零。

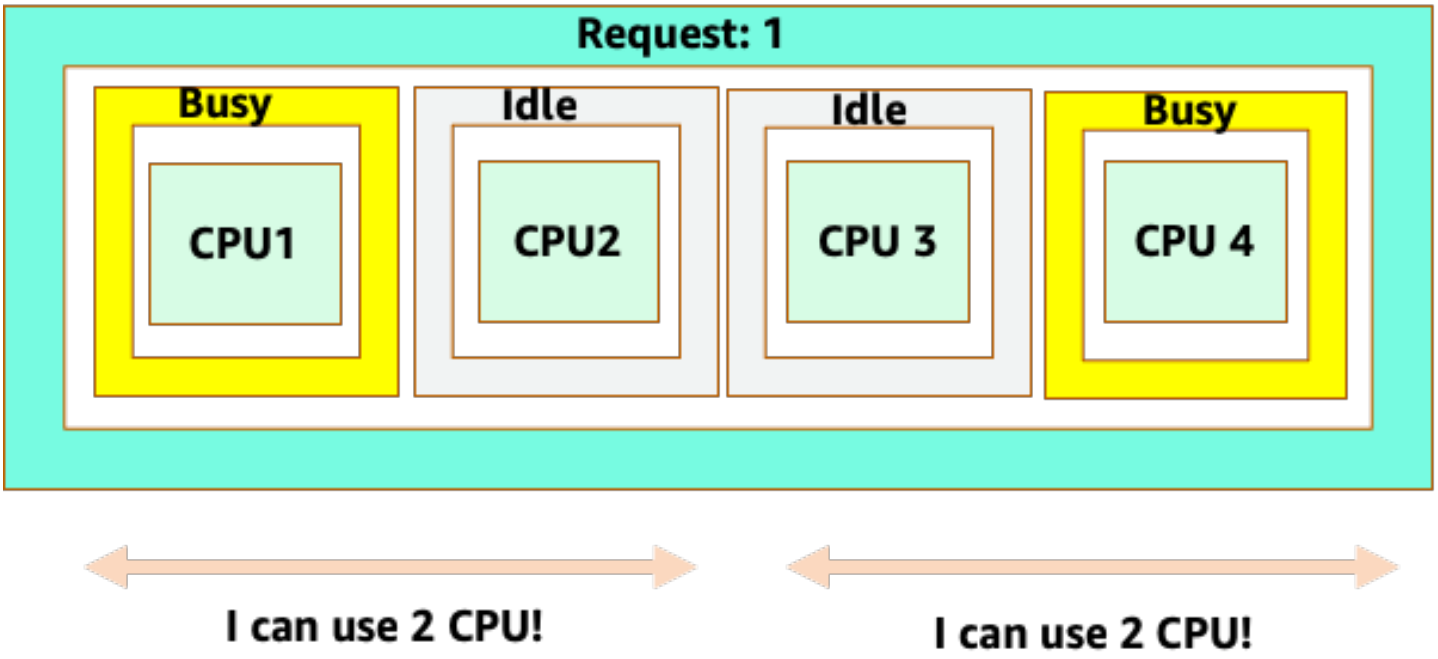


設定請求

將請求設定為該應用程式的 "sweet spot" 值會很有吸引力，但這會導致效率低下，如下圖所示。在這裡，我們已將請求值設定為 2 個 vCPU，但這些 Pod 的平均使用率大部分時間只會執行 1 個 CPU。此設定會導致我們浪費 50% 的 CPU 週期，這是無法接受的。



這可讓我們找到問題的複雜答案。在清空中無法考慮容器使用率；一個必須考慮節點上執行的其他應用程式。在下列範例中，本質上爆量的容器會與兩個低 CPU 使用率容器混合，這些容器可能受到記憶體限制。以這種方式，我們允許容器到達他們的甜點，而無需對節點課稅。



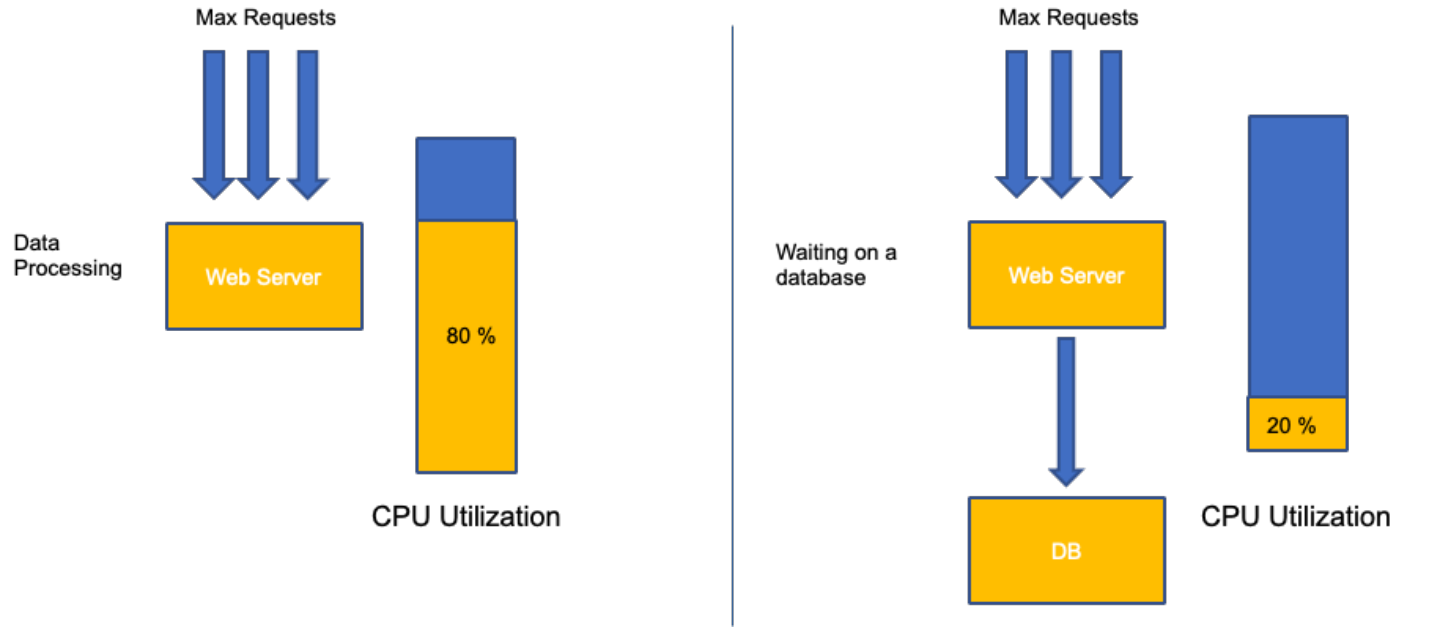
要從中採取的重要概念是，使用核心的 Kubernetes 排程器概念來了解 Linux 容器效能可能會導致決策不佳，因為它們無關。

> **Note**
>
> Linux MCU 有其強而有力的點。對於 I/O 型工作負載尤其如此。不過，如果您的應用程式使用不含附屬項目的完整核心，而且沒有 I/O 需求，CPU 鎖定可以從此程序中移除很大的複雜性，並受到這些警告的鼓勵。

使用率與飽和度

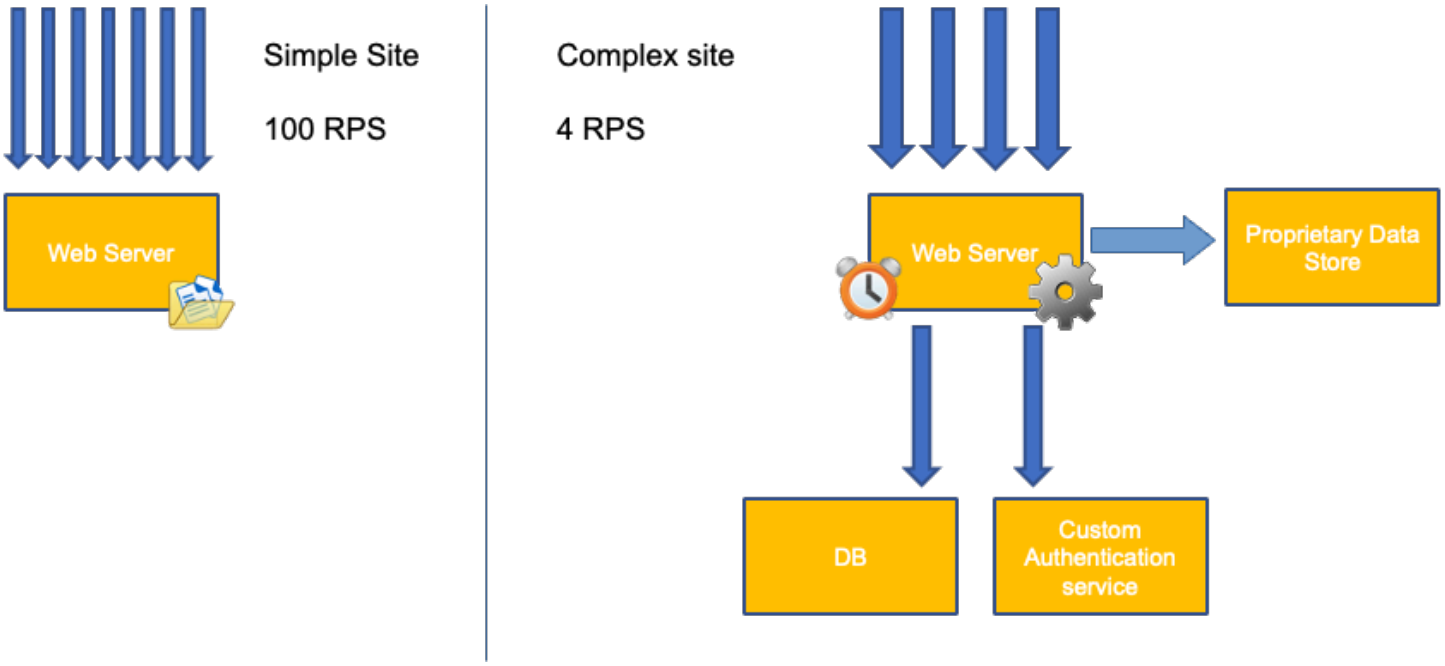
應用程式擴展的常見錯誤是僅使用擴展指標的 CPU 使用率。在複雜的應用程式中，這幾乎一律是應用程式實際上因請求而飽和的不良指標。在左側範例中，我們看到所有請求實際上都在命中 Web 伺服器，因此 CPU 使用率在飽和的情況下追蹤良好。

在真實世界的應用程式中，其中一些請求可能會由資料庫層或身分驗證層等提供服務。在這種情況下，請注意，CPU 不會在飽和的情況下追蹤，因為請求是由其他實體提供服務。在這種情況下，CPU 是飽和的極差指標。



在應用程式效能中使用錯誤的指標，是 Kubernetes 中不必要且無法預測擴展的首要原因。為您正在使用的應用程式類型挑選正確的飽和指標時，必須特別小心。請務必注意，沒有一個大小適合所有可以提供的建議。根據所使用的語言和有問題的應用程式類型，有一組不同的飽和度指標。

我們可能會認為此問題僅與 CPU 使用率有關，但其他常見的指標，例如每秒請求數，也可能遇到與上述完全相同的問題。請注意，請求也可以移至資料庫層、驗證層，而不是由我們的 Web 伺服器直接服務，因此 Web 伺服器本身的真實飽和指標很差。



遺憾的是，在挑選正確的飽和指標時，沒有簡單的答案。以下是一些需要考慮的指導方針：

- 了解您的語言執行時間 - 具有多個作業系統執行緒的語言與單一執行緒應用程式的反應不同，因此對節點的影響不同。
- 了解正確的垂直擴展 - 在擴展新 Pod 之前，您希望應用程式垂直擴展中有多少緩衝？
- 哪些指標真正反映應用程式的飽和 - Kafka Producer 的飽和指標與複雜的 Web 應用程式截然不同。
- 節點上的所有其他應用程式如何互相影響 - 在清空節點上的其他工作負載時，應用程式效能不會完成。

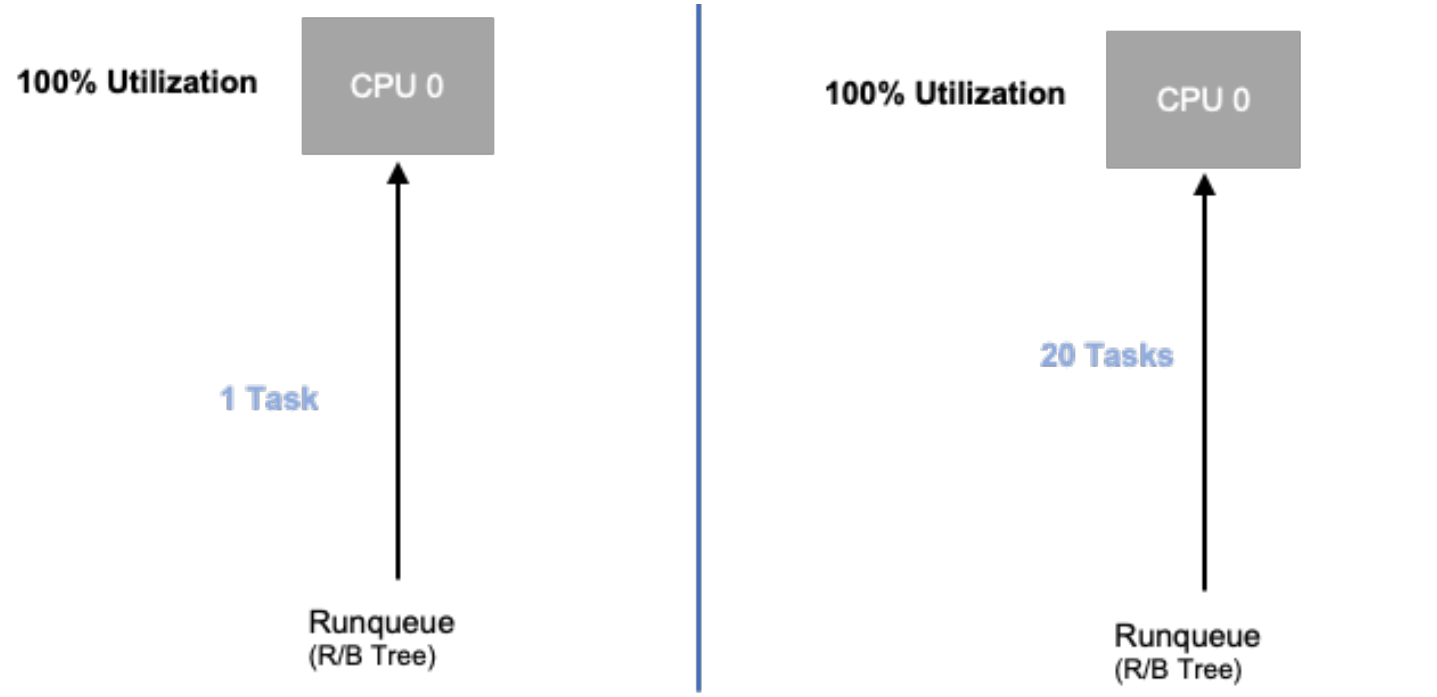
若要結束本節，可以很容易地將上述內容關閉為過於複雜且不必要的。我們通常會遇到問題，但我們不知道問題的真正本質，因為我們看到錯誤的指標。在下一節中，我們將探討發生這種情況的方式。

節點飽和度

現在我們已經探索應用程式飽和，讓我們從節點的觀點來看這個相同的概念。讓我們採用 100% 使用的兩個 CPUs，以查看使用率與飽和度之間的差異。

左側的 vCPU 是 100% 使用的，但沒有其他任務正在等待在此 vCPU 上執行，因此從理論上來說，這相當有效率。同時，我們在第二個範例中有 20 個單一執行緒應用程式等待 vCPU 處理。現在，所有 20 個應用程式在等待 vCPU 處理時都會遇到某種類型的延遲。換句話說，右側的 vCPU 是飽和的。

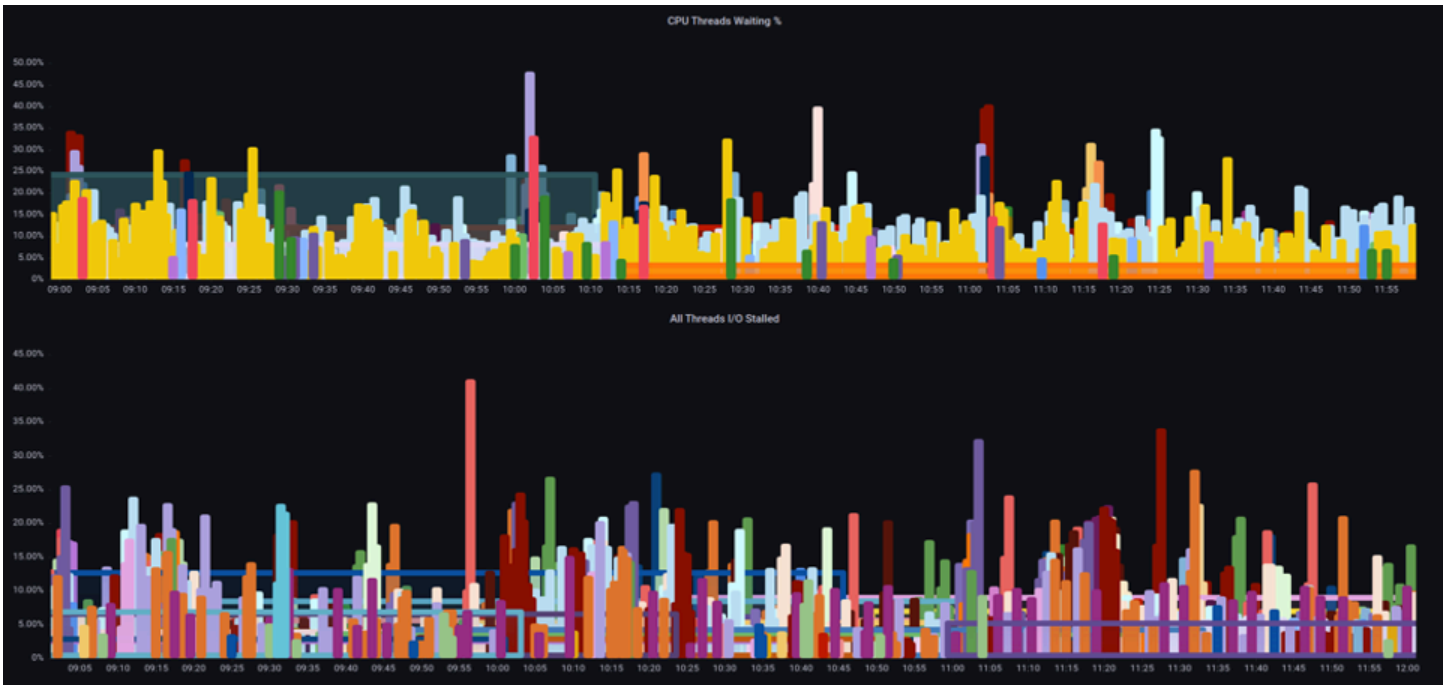
只有當我們只查看使用率時，我們才會看到此問題，但我們可能會將此延遲歸因於無關的內容，例如聯網，這會導致我們走錯路徑。



重要的是檢視飽和指標，而不只是在增加任何指定時間在節點上執行的 Pod 總數時的使用率指標，因為我們可以輕易地錯過我們有過度飽和節點的事實。對於此任務，我們可以使用壓力停止資訊指標，如下圖所示。

PromQL - 停滯的 I/O

```
topk(3, ((irate(node_pressure_io_stalled_seconds_total[1m])) * 100))
```



> **Note**
>
> 如需壓力停止指標的詳細資訊，請參閱 https：//https://facebookmicrosites.github.io/psi/docs/overview*

透過這些指標，我們可以判斷執行緒是否在 CPU 上等待，或即使盒上的每個執行緒都停滯在等待資源，例如記憶體或 I/O。 例如，我們可以看到執行個體上每個執行緒在 1 分鐘內停滯等待輸入/輸出的百分比。

```
topk(3, ((irate(node_pressure_io_stalled_seconds_total[1m])) * 100))
```

使用此指標，我們可以在上圖中看到盒子上的每個執行緒停滯了 45% 在高水位處等待輸入/輸出的時間，這表示我們在該分鐘拋棄了所有這些 CPU 週期。了解這種情況發生有助於我們回收大量的 vCPU 時間，從而提高擴展效率。

HPA V2

建議使用 HPA API 的 autoscaling/v2 版本。在某些情況下，較舊版本的 HPA API 可能會卡住擴展。它還僅限於在每個擴展步驟期間加倍的 Pod，這會為需要快速擴展的小型部署產生問題。

Autoscaling/v2 可讓我們更靈活地包含多個擴展條件，並在使用自訂和外部指標（非 K8s 指標）時，讓我們有極大的靈活性。

例如，我們可以擴展最高三個值（請參閱以下內容）。如果所有 Pod 的平均使用率超過 50%、如果自訂指標每秒輸入封包超過平均 1，000，或輸入物件超過每秒 10K000 個請求，我們會進行擴展。

> **Note**
>
> 這只是為了顯示自動擴展 API 的彈性，我們建議避免在生產環境中難以進行故障診斷的過於複雜的規則。

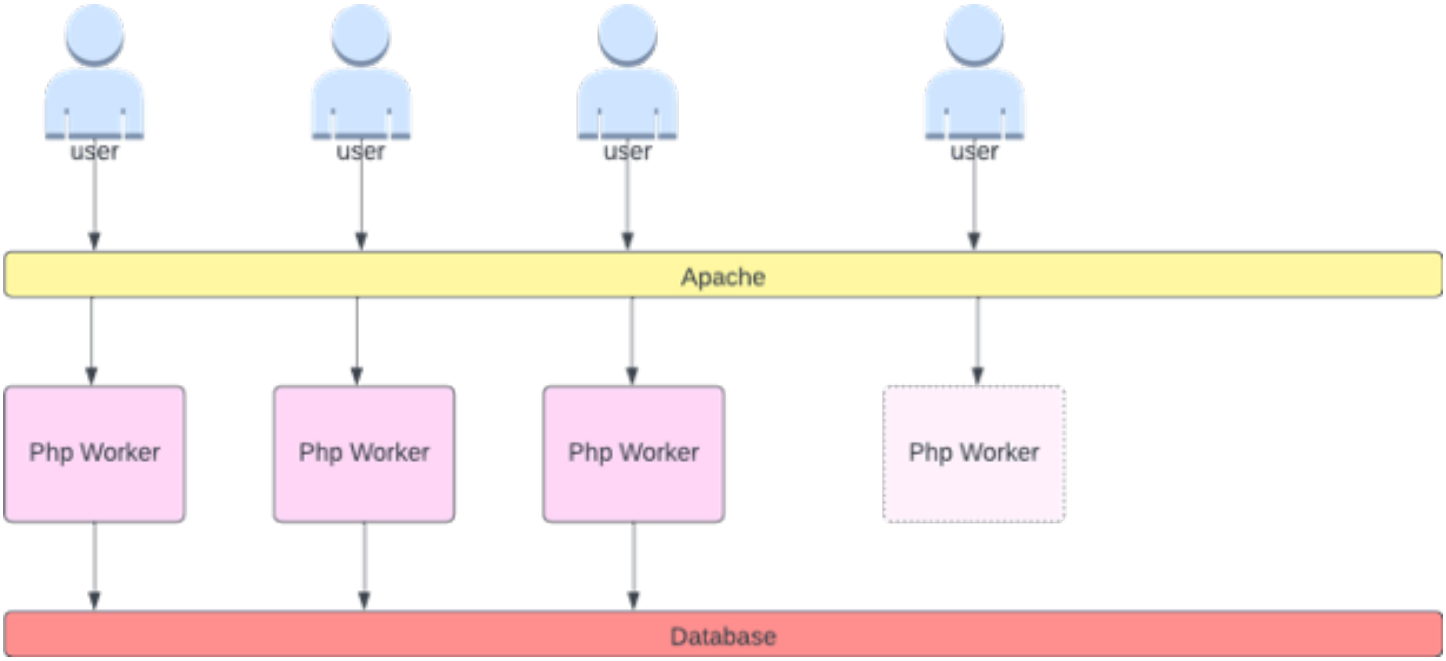
```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
  - type: Pods
    pods:
```

```
    metric:
      name: packets-per-second
    target:
      type: AverageValue
      averageValue: 1k
- type: Object
  object:
    metric:
      name: requests-per-second
    describedObject:
      apiVersion: networking.k8s.io/v1
      kind: Ingress
      name: main-route
    target:
      type: Value
      value: 10k
```

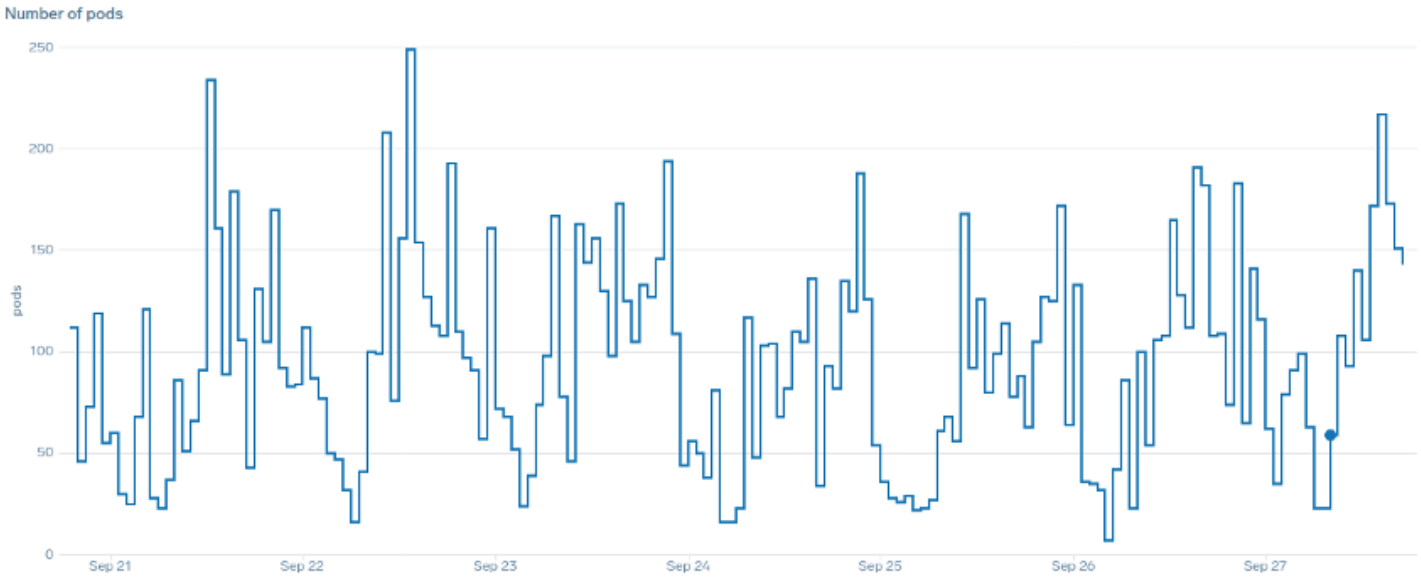
不過，我們了解了將此類指標用於複雜 Web 應用程式的風險。在這種情況下，我們會使用自訂或外部指標，準確反映應用程式飽和與使用率，以提供更好的服務。HPAv2 能夠根據任何指標進行擴展，從而實現這一點，但我們仍然需要找到該指標並將其匯出到 Kubernetes 以供使用。

例如，我們可以查看 Apache 中的作用中執行緒佇列計數。這通常會建立「更順暢」的擴展描述檔（該術語很快就會更詳細）。如果執行緒處於作用中狀態，則該執行緒是在資料庫層上等待還是在本機服務請求並不重要，如果使用所有應用程式執行緒，則表示應用程式已飽和。

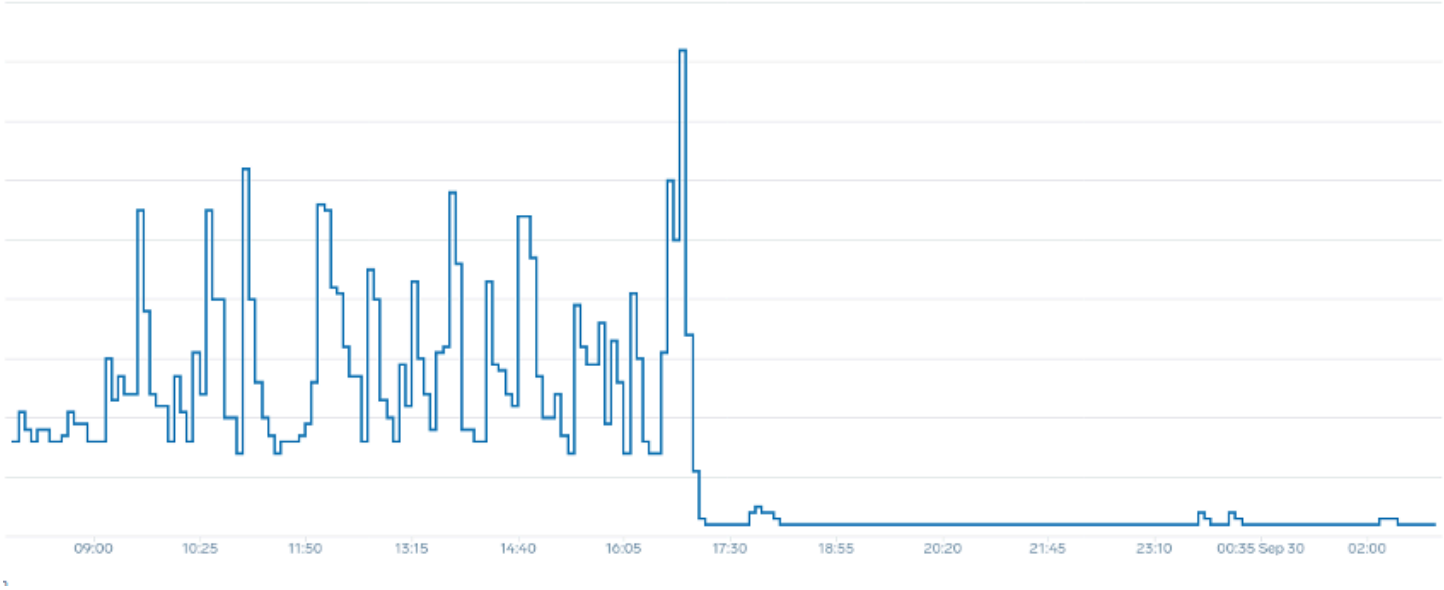
我們可以使用此執行緒耗盡做為訊號，以建立具有完整可用執行緒集區的新 Pod。這也讓我們能夠控制在大量流量期間，我們希望應用程式中的緩衝區吸收多大。例如，如果總執行緒集區為 10，則擴展 4 個使用的執行緒與 8 個使用的執行緒會對擴展應用程式時可用的緩衝產生重大影響。設定 4 對需要在繁重負載下快速擴展的應用程式來說很有意義，其中設定 8 會更有效率地使用我們的資源，如果我們有足夠的時間因為請求數量隨著時間的推移緩慢增加和急遽增加而擴展。



在擴展方面，「平滑」一詞是什麼意思？ 請注意下表，我們使用 CPU 做為指標。此部署中的 Pod 會在短時間內激增，從 50 個 Pod 開始，最多只有 250 個 Pod 才能立即縮減規模。這是叢集上 churn 的主要原因，是高效率擴展。



請注意，在我們變更為反映應用程式正確甜點（圖表中部）的指標後，如何順利擴展。我們的擴展現在是有效率的，並且我們的 Pod 可以透過調整請求設定來完全擴展我們提供的前端空間。現在，一小群 Pod 正在執行數百個 Pod 之前執行的工作。真實世界資料顯示，這是 Kubernetes 叢集可擴展性的第一大因素。

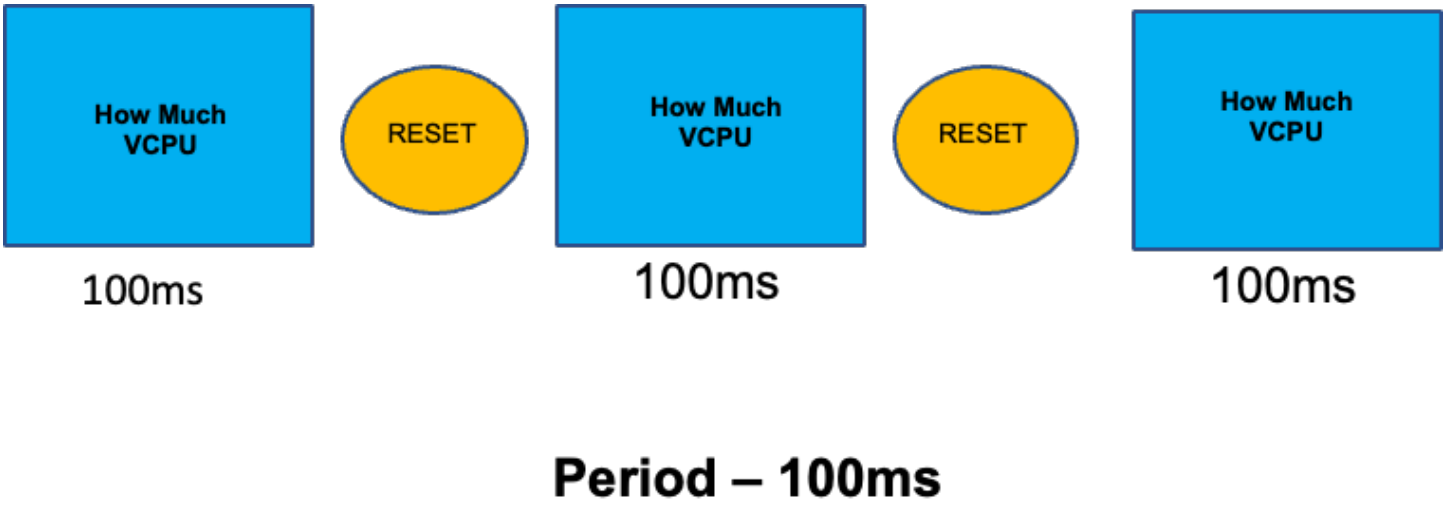


關鍵重點是 CPU 使用率只是應用程式和節點效能的一個維度。使用 CPU 使用率做為節點和應用程式的唯一運作狀態指標，會造成擴展、效能和成本方面的問題，這些都是緊密連結的概念。應用程式和節點的效能越高，您需要的擴展就越少，進而降低成本。

尋找和使用正確的飽和指標來擴展特定應用程式，也可讓您監控和警示該應用程式的真正瓶頸。如果略過此關鍵步驟，則報告效能問題會難以理解，如果不是不可能的話。

設定 CPU 限制

為了更完整地說明誤會的主題，我們將介紹 CPU 限制。簡言之，限制是與容器相關聯的中繼資料，容器具有每 100 毫秒重設一次的計數器。這有助於 Linux 追蹤特定容器在 100 毫秒期間內在節點範圍內使用多少 CPU 資源。



設定限制的常見錯誤是假設應用程式為單一執行緒，且僅在其「`assigned」` vCPU 上執行。在上一節中，我們了解到，CFS 不會指派核心，實際上，執行大型執行緒集區的容器會排程在盒上的所有可用 vCPU。

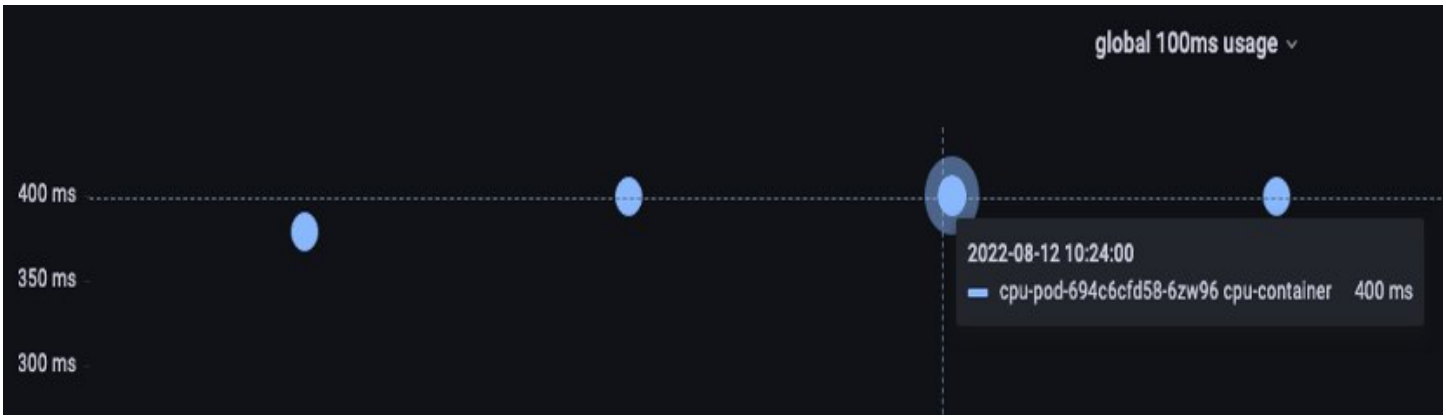
如果 64 個作業系統執行緒跨 64 個可用核心執行（從 Linux 節點的角度來看），我們將在所有 64 個核心上執行的時間加總後，讓 100 毫秒期間內使用過 CPU 時間的總帳單變得相當大。由於這可能只會在垃圾回收程序期間發生，因此很容易錯過這類情況。這就是為什麼在嘗試設定限制之前，必須使用指標來確保一段時間內的正確用量。

幸運的是，我們有一個方法可以查看應用程式中所有執行緒正在使用多少 vCPU。我們將`container_cpu_usage_seconds_total`為此目的使用 指標。

由於限流邏輯每 100 毫秒發生一次，而此指標是每秒指標，我們將 PromQL 以符合此 100 毫秒期間。如果您想要深入了解此 PromQL 陳述式工作，請參閱下列部落格。

PromQL 查詢：

```
topk(3, max by (pod, container)(rate(container_cpu_usage_seconds_total{image!="",
 instance="$instance"}[$__rate_interval]))) / 10
```



一旦我們認為我們擁有正確的價值，就可以在生產環境中設定限制。然後，需要查看我們的應用程式是否因為意外狀況而受到調節。我們可以透過查看 `container_cpu_throttled_seconds_total`

```
topk(3, max by (pod, container)(rate(container_cpu_cfs_throttled_seconds_total{image!
=``""``, instance=``"$instance"``}[$__rate_interval]))) / 10
```

記憶體

記憶體配置是另一個範例，其中很容易混淆 Linux CGroup 行為的 Kubernetes 排程行為。這是更細微的主題，因為 CGroup v2 在 Linux 和 Kubernetes 中處理記憶體的方式發生重大變更，而 Kubernetes 已變更其語法以反映這一點；如需更多詳細資訊，請參閱此部落格。

與 CPU 請求不同，記憶體請求會在排程程序完成後未使用。這是因為我們無法像使用 CPU 一樣壓縮 CGroup v1 中的記憶體。這讓我們只具有記憶體限制，其旨在完全終止 Pod 以作為記憶體洩漏的故障安全。這是全部或部分的樣式主張，但現在我們已獲得解決此問題的新方法。

首先，請務必了解，為容器設定正確數量的記憶體並不簡單。Linux 中的檔案系統將使用記憶體做為快取，以改善效能。此快取會隨著時間成長，而且很難知道有多少記憶體適合快取，但可以回收，而不會對應用程式效能造成重大影響。這通常會導致錯誤解譯記憶體用量。

具備「壓縮」記憶體的能力是 CGroup v2 背後的主要驅動因素之一。如需有關為何需要 CGroup V2 的更多歷史記錄，請參閱 Chris Down 在 LISA21 的簡報，其中說明了為什麼無法正確設定最小記憶體是促使他建立 CGroup v2 和壓力停止指標的原因之一。

所幸，Kubernetes 現在的概念為 `memory.min`和 `memory.high`下`requests.memory`。這讓我們可以選擇積極釋出此快取記憶體，以供其他容器使用。一旦容器達到記憶體上限，核心就可以積極回收該容器的記憶體，直到 設定的值為止`memory.min`。因此，當節點受到記憶體壓力時，可提供更多彈性。

關鍵問題會變成要`memory.min`設為哪個值？ 這是記憶體壓力停滯指標發揮作用的地方。我們可以使用這些指標來偵測容器層級的記憶體「當機」。然後，我們可以使用 fbtax 等控制器，`memory.min`透過尋找此記憶體中斷來偵測 的正確值，並動態將`memory.min`值設定為此設定。

摘要

若要總結 區段，您可以輕鬆整合下列概念：

- 使用率和飽和度
- 具有 Kubernetes Scheduler 邏輯的 Linux 效能規則

必須特別小心，將這些概念分開。效能和擴展會在深度層級上連結。不必要的擴展會產生效能問題，進而產生擴展問題。

Kubernetes 上游 SLOs

Amazon EKS 執行與上游 Kubernetes 版本相同的程式碼，並確保 EKS 叢集在 Kubernetes 社群定義的 SLOs 內運作。Kubernetes Scalability Special Interest Group (SIG) 定義可擴展性目標，並透過 SLIs 和 SLOs 調查效能的瓶頸。

SLIs 是我們測量系統的方式，例如可用來判斷系統執行狀況的指標或指標，例如請求延遲或計數。SLOs會定義系統執行「良好」時預期的值，例如，請求延遲保持不到 3 秒。Kubernetes SLOs 和 SLIs 專注於 Kubernetes 元件的效能，並與專注於 EKS 叢集端點可用性的 Amazon EKS 服務 SLAs 完全獨立。

Kubernetes 有許多功能，可讓使用者使用自訂附加元件或驅動程式來擴展系統，例如 CSI 驅動程式、許可 Webhook 和自動擴展器。這些擴充功能可能會以不同方式大幅影響 Kubernetes 叢集的效能，例如，如果 Webhook 目標無法使用，則使用 的許可 Webhook `failurePolicy=Ignore`可能會為 K8s API 請求增加延遲。Kubernetes 可擴展性 SIG 使用「您承諾，我們承諾」架構來定義可擴展性：

> 如果您承諾：- 正確設定叢集 - 使用擴充功能「合理」- 將叢集中的負載保持在建議限制內
>
> 然後我們承諾您的叢集會擴展，即：- 所有 SLOs都滿足。

Kubernetes SLOs

Kubernetes SLOs 不會考慮可能影響叢集的所有外掛程式和外部限制，例如工作者節點擴展或許可 Webhook。這些 SLOs著重於 Kubernetes 元件，並確保 Kubernetes 動作和資源在預期範圍內運作。SLOs 可協助 Kubernetes 開發人員確保 Kubernetes 程式碼的變更不會降低整個系統的效能。

Kuberntes 可擴展性 SIG 定義了下列官方 SLO/SLIs。Amazon EKS 團隊會定期對這些 SLOs/SLIs 的 EKS 叢集執行可擴展性測試，以在進行變更並發佈新版本時監控效能降低。

目標	定義	SLO
API 請求延遲（變動）	處理每個（資源、動詞）對之單一物件變動 API 呼叫的延遲，測量為過去 5 分鐘內的第 99 個百分位數	在預設 Kubernetes 安裝中，針對每個（資源、動詞）配對，不包括虛擬和彙總資源和自訂資源定義，每個叢集天的第 99 個百分位數 <= 1 秒
API 請求延遲（唯讀）	處理每個（資源、範圍）對非串流唯讀 API 呼叫的延遲，測量為過去 5 分鐘內的第 99 個百分位數	在預設 Kubernetes 安裝中，對於每個（資源、範圍）對，不包括虛擬和彙總資源和自訂資源定義，每個叢集日的第 99 個百分位數：(a) <= 1s，如果 `scope=resource` (b) <= 30s（如果 `scope=namespace` 或 `scope=cluster`）
Pod 啟動延遲	可排程無狀態 Pod 的啟動延遲，不包括提取映像和執行初始化容器的時間，從 Pod 建立時間戳記到透過觀察報告其所有容器為啟動和觀察的時間，測量為過去 5 分鐘內的第 99 個百分位數	在預設 Kubernetes 安裝中，每個叢集天的第 99 個百分位數 <= 5s

API 請求延遲

根據`1m0s`預設，`kube-apiserver` 已`--request-timeout`定義為 ，這表示請求在逾時和取消之前最多可以執行一分鐘 (60 秒）。為延遲定義的 SLOs 會依所提出的請求類型細分，可以是變動或唯讀：

變動

在 Kubernetes 中變更請求會變更資源，例如建立、刪除或更新。這些請求非常昂貴，因為在傳回更新後的物件之前，必須將這些變更寫入至 等後端。Etcd 是分散式金鑰值存放區，用於所有 Kubernetes 叢集資料。

對於（資源、動詞）對 Kubernetes 資源，此延遲測量為 5 分鐘內的第 99 個百分位數，例如，這會測量建立 Pod 請求和更新節點請求的延遲。請求延遲必須 <= 1 秒才能滿足 SLO。

唯讀

唯讀請求會擷取單一資源（例如取得 Pod X) 或集合（例如「從命名空間 X 取得所有 Pod」)。會kube-apiserver維護物件的快取，因此可能會從快取傳回請求的資源，或者可能需要先從 等項目擷取它們。這些延遲也會由 5 分鐘內的第 99 個百分位數測量，但唯讀請求可以有不同的範圍。SLO 定義了兩個不同的目標：

- 對於對單一資源提出的請求（即 `kubectl get pod -n mynamespace my-controller-xxx`），請求延遲應保持 <= 1 秒。
- 對於對命名空間或叢集中的多個資源提出的請求（例如 `kubectl get pods -A`)，延遲應保持 <= 30 秒

SLO 對不同的請求範圍具有不同的目標值，因為對 Kubernetes 資源清單提出的請求預期請求中所有物件的詳細資訊會在 SLO 中傳回。在大型叢集或大型資源集合上，這可能會導致大型回應大小，這可能需要一些時間才能傳回。例如，在執行數萬個 Pod 的叢集中，當以 JSON 編碼時，每個 Pod 大約為 1 KiB，傳回叢集中的所有 Pod 會包含 10MB 或更多。Kubernetes 用戶端可以使用 APIListChunking 來擷取大量資源集合，協助減少此回應大小。

Pod 啟動延遲

此 SLO 主要關注從建立 Pod 到該 Pod 中的容器實際開始執行所需的時間。為了測量此值與 Pod 上記錄的建立時間戳記的差異，以及當該 Pod 上的 WATCH 報告已開始計算容器時（不包括容器映像提取和初始化容器執行的時間）。為了滿足 SLO，此 Pod 啟動延遲的每個叢集天的第 99 個百分位數必須保持 <=5 秒。

請注意，此 SLO 假設此叢集中已存在工作者節點處於就緒狀態，以便排定 Pod。此 SLO 不會考慮映像提取或初始化容器執行，也限制測試為「無狀態 Pod」，其不會利用持久性儲存外掛程式。

Kubernetes SLI 指標

Kubernetes 也正在改善 SLIs 的可觀測性，方法是將 Prometheus 指標新增至追蹤這些 SLIs 一段時間的 Kubernetes 元件。使用 Prometheus 查詢語言 (PromQL)，我們可以建置在 Prometheus 或 Grafana 儀表板等工具中顯示 SLI 效能的查詢，以下是上述 SLOs 的一些範例。

API 伺服器請求延遲

指標	定義
apiserver_request_sli_duration_seconds	每個動詞、群組、版本、資源、子資源、範圍和元件的回應延遲分佈（未計算 Webhook 持續時間和優先順序和公平性佇列等待時間），以秒為單位。
apiserver_request_duration_seconds	每個動詞、試轉值、群組、版本、資源、子資源、範圍和元件的回應延遲分佈，以秒為單位。

> **Note**
>
> 從 Kubernetes 1.27 開始，即可使用 `apiserver_request_sli_duration_seconds` 指標。

您可以使用這些指標來調查 API Server 回應時間，以及 Kubernetes 元件或其他外掛程式/元件是否有瓶頸。以下查詢是以社群 SLO 儀表板為基礎。

API 請求延遲 SLI（變動）- 這次不包含 Webhook 執行或佇列中等待的時間。

```
histogram_quantile(0.99,
sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"CREATE|
DELETE|PATCH|POST|PUT", subresource!~"proxy|attach|log|exec|portforward"}
[5m])) by (resource, subresource, verb, scope, le)) > 0
```

API 請求延遲總計（變動）- 這是請求在 API 伺服器上接受的總時間，這次可能比 SLI 時間長，因為它包括 Webhook 執行和 API 優先順序和公平性等待時間。

```
histogram_quantile(0.99,
sum(rate(apiserver_request_duration_seconds_bucket{verb=~"CREATE|DELETE|
PATCH|POST|PUT", subresource!~"proxy|attach|log|exec|portforward"}[5m])) by
(resource, subresource, verb, scope, le)) > 0
```

在這些查詢中，我們會排除未立即傳回的串流 API 請求，例如 `kubectl port-forward`或 `kubectl exec`請求 (`subresource!~"proxy|attach|log|exec|portforward"`)，而且只會篩選修改物件 () 的 Kubernetes 動詞`verb=~"CREATE|DELETE|PATCH|POST|PUT"`。然後，我們將計算過去 5 分鐘內該延遲的第 99 個百分位數。

我們可以對唯讀 API 請求使用類似的查詢，只需修改我們正在篩選的動詞，以包含唯讀動作 LIST和 GET。根據請求的範圍，也有不同的 SLO 閾值，即取得單一資源或列出一些資源。

API 請求延遲 SLI （唯讀） - 這次不包含 Webhook 執行或佇列中等待的時間。對於單一資源 (scope=resource， threshold=1s) `histogram_quantile(0.99, sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"GET", scope=~"resource"}[5m])) by (resource, subresource, verb, scope, le))`

對於相同命名空間中的資源集合 (scope=namespace， threshold=5s) `histogram_quantile(0.99, sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"LIST", scope=~"namespace"}[5m])) by (resource, subresource, verb, scope, le))`

對於整個叢集的資源集合 (scope=cluster， threshold=30s) `histogram_quantile(0.99, sum(rate(apiserver_request_sli_duration_seconds_bucket{verb=~"LIST", scope=~"cluster"}[5m])) by (resource, subresource, verb, scope, le))`

API 請求延遲總計 （唯讀） - 這是請求在 API 伺服器上接受的總時間，這次可能比 SLI 時間更長，因為它包含 Webhook 執行和等待時間。對於單一資源 (scope=resource， threshold=1s) `histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"GET", scope=~"resource"}[5m])) by (resource, subresource, verb, scope, le))`

對於相同命名空間中的資源集合 (scope=namespace， threshold=5s) `histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"LIST", scope=~"namespace"}[5m])) by (resource, subresource, verb, scope, le))`

對於整個叢集的資源集合 (scope=cluster， threshold=30s) `histogram_quantile(0.99, sum(rate(apiserver_request_duration_seconds_bucket{verb=~"LIST", scope=~"cluster"}[5m])) by (resource, subresource, verb, scope, le))`

SLI 指標透過排除請求在 API Priority and Fairness 佇列中等待的時間、透過許可 Webhook 或其他 Kubernetes 延伸模組運作，來深入了解 Kubernetes 元件的效能。總指標提供更全面的檢視，因為它反映您的應用程式等待 API 伺服器回應的時間。比較這些指標可以深入了解請求處理發生延遲的位置。

Pod 啟動延遲

指標	定義
kubelet_pod_start_sli_duration_seconds	以秒為單位啟動 Pod 的持續時間，不包括提取影像和執行初始化容器的時間，從 Pod 建立時間戳記開始測量，直到所有容器回報為透過監視啟動和觀察為止
kubelet_pod_start_duration_seconds	從第一次看到 Pod 的 kubelet 到開始執行 Pod 的持續時間，以秒為單位。這不包括排程 Pod 或向外擴展工作者節點容量的時間。

> **Note**
>
> kubelet_pod_start_sli_duration_seconds 從 Kubernetes 1.27 開始提供。

與上述查詢類似，您可以使用這些指標來深入了解與 Kubelet 動作相比，節點擴展、映像提取和初始化容器延遲 Pod 啟動的時間。

Pod 啟動延遲 SLI - 這是從建立 Pod 到應用程式容器回報為執行中的時間。這包括工作者節點容量可用和 Pod 排程所需的時間，但這不包括提取映像或初始化容器執行所需的時間。 `histogram_quantile(0.99, sum(rate(kubelet_pod_start_sli_duration_seconds_bucket[5m])) by (le))`

Pod 啟動延遲總計 - 這是第一次啟動 Pod 所需的時間。這是從 kubelet 透過 WATCH 接收 Pod 時開始測量，其中不包含工作者節點擴展或排程的時間。這包括提取映像和初始化容器以執行的時間。 `histogram_quantile(0.99, sum(rate(kubelet_pod_start_duration_seconds_bucket[5m])) by (le))`

叢集上的 SLOs

如果您要從 EKS 叢集中的 Kubernetes 資源收集 Prometheus 指標，您可以深入了解 Kubernetes 控制平面元件的效能。

效能測試儲存庫包含 Grafana 儀表板，可在測試期間顯示叢集的延遲和關鍵效能指標。效能測試組態利用 kube-prometheus-stack，這是設定為收集 Kubernetes 指標的開放原始碼專案，但您也可以使用 Amazon Managed Prometheus 和 Amazon Managed Grafana。

如果您使用的是 kube-prometheus-stack或類似的 Prometheus 解決方案，您可以安裝相同的儀表板，以即時觀察叢集上的 SLOs。

1. 您必須先安裝在具有 的儀表板中使用的 Prometheus 規則kubectl apply -f prometheus-rules.yaml。您可以在這裡下載規則的副本：https：//https://github.com/kubernetes/perf-tests/blob/master/clusterloader2/pkg/prometheus/manifests/prometheus-rules.yaml
   a. 請務必檢查 檔案中的命名空間是否符合您的環境
   b. 如果您使用 ，請確認標籤符合 prometheus.prometheusSpec.ruleSelector helm 值 kube-prometheus-stack
2. 然後，您可以在 Grafana 中安裝儀表板。此處提供用於產生它們的 json 儀表板和 python 指令碼：https：//https://github.com/kubernetes/perf-tests/tree/master/clusterloader2/pkg/prometheus/manifests/dashboards
   a. slo.json 儀表板會顯示叢集與 Kubernetes SLOs 相關的效能

假設 SLOs專注於叢集中 Kubernetes 元件的效能，但您可以檢閱其他指標，為您的叢集提供不同的觀點或洞見。Kubernetes 社群專案，例如 Kube-state-metrics，可協助您快速分析叢集中的趨勢。Kubernetes 社群的最常見外掛程式和驅動程式也會發出 Prometheus 指標，讓您調查自動擴展器或自訂排程器等項目。

可觀測性最佳實務指南提供其他 Kubernetes 指標的範例，您可以用來深入了解。

已知限制Service Quotas

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

Amazon EKS 可用於各種工作負載，並且可以與各種 AWS 服務互動，我們看到客戶工作負載遇到類似的 AWS 服務配額和其他阻礙可擴展性的問題。

您的 AWS 帳戶具有預設配額 （您的團隊可請求的每個 AWS 資源數量上限）。每個 AWS 服務都會定義自己的配額，而配額通常是區域特定的。您可以請求提高某些配額 （軟性限制），而其他配額無法

提高（硬性限制）。在建構應用程式時，您應該考慮這些值。請考慮定期檢閱這些服務限制，並將其納入您的應用程式設計中。

您可以檢閱帳戶中的用量，並在 AWS Service Quotas 主控台或使用 AWS CLI 開啟配額增加請求。如需 Service Quotas 的詳細資訊，以及任何進一步的限制或增加通知，請參閱個別 AWS Service 的 AWS 文件。

> **Note**
>
> Amazon EKS Service Quotas 會列出服務配額，並在可用時提供請求增加的連結。

其他 AWS Service Quotas

我們看到 EKS 客戶受到下列其他 AWS 服務配額的影響。其中一些可能僅適用於特定使用案例或組態，但您可能會考慮您的解決方案是否會在擴展時遇到任何這些情況。配額由 Service 組織，每個配額都有 L-XXXXXXXX 格式的 ID，可用於在 AWS Service Quotas 主控台中查詢

服務	配額 (L-xxxxx)	影響	ID (L-xxxxx)	預設
IAM	每個帳戶的角色	可以限制帳戶中的叢集或 IRSA 角色數量。	L-FE177D64	1,000
IAM	每個帳戶的 OpenId 連線供應商	可以限制每個帳戶的叢集數量，IRSA 會使用 OpenID Connect	L-858F3967	100
IAM	角色信任政策長度	可以限制 IAM 角色與 IRSA 相關聯的叢集數量	L-C07B4B0D	2,048
VPC	每個網路介面的安全群組數	可以限制叢集聯網的控制或連線能力	L-2AFB9258	5

服務	配額 (L-xxxxx)	影響	ID (L-xxxxx)	預設
VPC	每個 VPC 的 IPv4 CIDR 區塊數	可以限制 EKS 工作者節點的數量	L-83CA0A9D	5
VPC	每個路由表的路由數	可以限制叢集聯網的控制或連線能力	L-93826ACB	50
VPC	每個 VPC 的作用中 VPC 對等連接數	可以限制叢集聯網的控制或連線能力	L-7E9ECCDB	50
VPC	每個安全群組的傳入或傳出規則。	可以限制叢集聯網的控制或連線能力，EKS 中的某些控制器會建立新的規則	L-0EA8095F	50
VPC	每個區域的 VPC 數	可以限制每個帳戶的叢集數量，或叢集聯網的控制或連線能力	L-F678F1CE	5
VPC	每個區域的網際網路閘道數	可以限制每個帳戶的叢集數量，或叢集聯網的控制或連線能力	L-A4707A72	5
VPC	每個區域的網路介面	可以限制 EKS 工作者節點的數量，或影響 EKS 控制平面擴展/更新活動。	L-DF5E4CA3	5,000

服務	配額 (L-xxxxx)	影響	ID (L-xxxxx)	預設
VPC	網路地址使用	可以限制每個帳戶的叢集數量，或叢集聯網的控制或連線能力	L-BB24F6E5	64，000
VPC	對等網路地址使用	可以限制每個帳戶的叢集數量，或叢集聯網的控制或連線能力	L-CD17FD4B	128，000
ELB	每個 Network Load Balancer 接聽程式	可以限制流量傳入至叢集的控制。	L-57A373D6	50
ELB	每個區域的目標群組	可以限制流量傳入至叢集的控制。	L-B22855CB	3,000
ELB	每個 Application Load Balancer 的目標	可以限制流量傳入至叢集的控制。	L-7E6692B2	1,000
ELB	每個 Network Load Balancer 目標	可以限制流量傳入至叢集的控制。	L-EEF1AD04	3,000
ELB	每個 Network Load Balancer 每個可用區域的目標	可以限制流量傳入至叢集的控制。	L-B211E961	500
ELB	每個區域每個目標群組的目標數	可以限制流量傳入至叢集的控制。	L-A0D0B863	1,000

服務	配額 (L-xxxxx)	影響	ID (L-xxxxx)	預設
ELB	每個區域的 Application Load Balancer	可以限制流量傳入至叢集的控制。	L-53DA6B97	50
ELB	每個區域的 Classic Load Balancer	可以限制流量傳入至叢集的控制。	L-E9E9831D	20
ELB	每個區域的網路負載平衡器	可以限制流量傳入至叢集的控制。	L-69A177A2	50
EC2	執行隨需標準 (A、C、D、H、I、M、R、T、Z) 執行個體（作為最大 vCPU 計數）	可以限制 EKS 工作者節點的數量	L-1216C47A	5
EC2	所有標準 (A、C、D、H、I、M、R、T、Z) Spot 執行個體請求（以 vCPU 計數上限表示）	可以限制 EKS 工作者節點的數量	L-34B43A08	5
EC2	EC2-VPC 彈性 IPs	可以限制 NAT GWs（以及 VPCs) 的數量，這可能限制區域中的叢集數量	L-0263D0A3	5
EBS	每個區域的快照	可以限制具狀態工作負載的備份策略	L-309BACF6	100,000

服務	配額 (L-xxxxx)	影響	ID (L-xxxxx)	預設
EBS	一般用途 SSD (gp3) 磁碟區的儲存體，以 TiB 為單位	可以限制 EKS 工作者節點或PersistentVolume儲存的數量	L-7A658B76	50
EBS	一般用途 SSD (gp2) 磁碟區的儲存體，以 TiB 為單位	可以限制 EKS 工作者節點或PersistentVolume儲存的數量	L-D18FCD1D	50
ECR	已註冊的儲存庫	可以限制叢集中的工作負載數量	L-CFEB8E8D	100,000
ECR	每個儲存庫的映像	可以限制叢集中的工作負載數量	L-03A36CE1	20,000
SecretsManager	每個區域的秘密	可以限制叢集中的工作負載數量	L-2F66C23C	500,000

AWS 請求調節

AWS 服務也會實作請求限流，以確保它們保持效能並可供所有客戶使用。與 Service Quotas 類似，每個 AWS 服務都會維護自己的請求限流閾值。如果您的工作負載需要快速發出大量 API 呼叫，或者您注意到應用程式中的請求限流錯誤，請考慮檢閱個別的 AWS 服務文件。

有關佈建 EC2 網路介面或 IP 地址的 EC2 API 請求可能會在大型叢集中遇到請求限流，或在叢集大幅擴展時遇到請求限流。下表顯示我們看到客戶遇到請求限流的一些 API 動作。您可以在速率調節的 EC2 文件中檢閱 EC2 速率限制預設值和請求提高速率限制的步驟。 EC2

變更動作	唯讀動作
AssignPrivateIpAddresses	DescribeDhcpOptions
AttachNetworkInterface	DescribeInstances
CreateNetworkInterface	DescribeNetworkInterfaces

變更動作	唯讀動作
DeleteNetworkInterface	DescribeSecurityGroups
DeleteTags	DescribeTags
DetachNetworkInterface	DescribeVpcs
ModifyNetworkInterfaceAttribute	DescribeVolumes
UnassignPrivateIpAddresses	

其他已知限制

- Route 53 也有對 Route 53 API 每秒 Route 53 個請求的極低速率限制。如果您有大量網域要使用外部 DNS 等專案進行更新，您可能會看到速率調節和更新網域的延遲。
  - 有些 Nitro 執行個體類型在 Amazon EBS 磁碟區、網路介面和 NVMe 執行個體存放區磁碟區之間共用的磁碟區連接限制為 28。 NVMe 如果您的工作負載掛載許多 EBS 磁碟區，您可能會遇到這些執行個體類型可達到的 Pod 密度限制
  - 每個 Ec2 執行個體可追蹤的最大連線數。如果您的工作負載正在處理大量連線，您可能會看到通訊失敗或錯誤，因為已達到此上限。您可以使用 `conntrack_allowance_available`和 `conntrack_allowance_exceeded` 網路效能指標來監控 EKS 工作者節點上追蹤的連線數目。
  - 在 EKS 環境中，根據上游指引，加密儲存限制為 8 GiB。請監控指標`apiserver_storage_size_bytes`以追蹤 db 大小。您可以參考警示規則`etcdBackendQuotaLowSpace`和 `etcdExcessiveDatabaseGrowth` 來設定此監控。

叢集升級的最佳實務

> **Tip**
>
> 透過 Amazon EKS 研討會探索最佳實務。

本指南示範叢集管理員如何規劃和執行其 Amazon EKS 升級策略。它還描述了如何升級自我管理節點、受管節點群組、Karpenter 節點和 Fargate 節點。它不包括 EKS Anywhere、自我管理 Kubernetes、AWS Outposts 或 AWS Local Zones 的指導。

概觀

Kubernetes 版本包含控制平面和資料平面。為了確保操作順暢，控制平面和資料平面都應執行相同的 Kubernetes 次要版本，例如 1.24。雖然 AWS 會管理和升級控制平面，但更新資料平面中的工作者節點是您的責任。

- 控制平面 — 控制平面的版本由 Kubernetes API 伺服器決定。在 Amazon EKS 叢集中，AWS 會負責管理此元件。控制平面升級可以透過 AWS API 啟動。
- 資料平面 — 資料平面版本與在個別節點上執行的 Kubelet 版本相關聯。在同一叢集中，可能會有節點執行不同的版本。您可以執行 來檢查所有節點的版本`kubectl get nodes`。

升級之前

如果您打算在 Amazon EKS 中升級 Kubernetes 版本，在開始升級之前，應該實施幾個重要的政策、工具和程序。

- 了解棄用政策 — 深入了解 Kubernetes 棄用政策的運作方式。請注意可能影響現有應用程式的任何近期變更。較新版本的 Kubernetes 通常會淘汰某些 APIs和功能，這可能會導致執行應用程式的問題。
- 檢閱 Kubernetes 變更日誌 — 徹底檢閱 Kubernetes 變更日誌與 Amazon EKS Kubernetes 版本，以了解對叢集的任何可能影響，例如可能影響工作負載的中斷變更。
- 評估叢集附加元件相容性 — 發佈新版本或將叢集更新為新的 Kubernetes 次要版本後，Amazon EKS 不會自動更新附加元件。檢閱更新附加元件以了解任何現有叢集附加元件與您要升級之叢集版本的相容性。

- 啟用控制平面記錄 — 啟用控制平面記錄，以擷取升級程序期間可能發生的日誌、錯誤或問題。請考慮檢閱這些日誌是否有任何異常。在非生產環境中測試叢集升級，或將自動化測試整合到您的持續整合工作流程中，以評估與應用程式、控制器和自訂整合的版本相容性。
- 探索 eksctl for Cluster Management — 考慮使用 eksctl 來管理您的 EKS 叢集。它可讓您立即更新控制平面、管理附加元件，以及處理工作者節點更新out-of-the-box。
- 選擇自動模式或受管節點群組 — 使用自動模式或 EKS 受管節點群組簡化和自動化工作者節點升級。這些選項可簡化程序並減少手動介入。
- 使用 kubectl 轉換外掛程式 — 利用 kubectl 轉換外掛程式來促進不同 API 版本之間的 Kubernetes 資訊清單檔案轉換。這有助於確保您的組態與新的 Kubernetes 版本相容。

讓您的叢集保持在up-to-date

保持最新的 Kubernetes 更新對於安全且有效率的 EKS 環境至關重要，反映 Amazon EKS 中的共同責任模型。透過將這些策略整合到您的操作工作流程中，您將自行定位，以維護up-to-date且安全的叢集，充分利用最新的功能和改進。策略：

- 支援的版本政策 — 與 Kubernetes 社群 保持一致，Amazon EKS 通常提供三個作用中的 Kubernetes 版本。Kubernetes 次要版本在 Amazon EKS 發行後的前 14 個月內都受到標準支援。一旦過了標準支援日期，後續 12 個月版本就會進入延長支援。棄用通知會在版本達到其標準支援結束日期的至少 60 天前發出。如需詳細資訊，請參閱 EKS 版本生命週期文件。
- 自動升級政策 — 我們強烈建議與 EKS 叢集中的 Kubernetes 更新保持同步。對於已完成 26 個月生命週期的 Kubernetes 版本，在其中執行的叢集 (14 個月的標準支援加 12 個月的延長支援) 將自動升級到下一版本。請注意，您可以停用延伸支援。無法在版本的end-of-life觸發自動升級之前主動升級，這可能會中斷您的工作負載和系統。如需詳細資訊，請參閱 EKS 版本FAQs。
- 建立升級 Runbook — 建立記錄良好的程序來管理升級。作為主動方法的一部分，開發專為升級程序量身打造的 Runbook 和專用工具。這不僅可以增強您的準備度，還可以簡化複雜的轉換。讓它成為標準做法，每年至少升級您的叢集一次。此實務可讓您符合持續的技術進展，進而提升您環境的效率和安全性。

檢閱 EKS 發行行事曆

檢閱 EKS Kubernetes 版本行事曆，以了解新版本何時推出，以及特定版本的支援何時結束。一般而言，EKS 每年發行三個次要版本的 Kubernetes，每個次要版本支援約 14 個月。

此外，請檢閱上游 Kubernetes 版本資訊。

了解共同責任模型如何套用至叢集升級

您負責啟動叢集控制平面和資料平面的升級。了解如何啟動升級。當您啟動叢集升級時，AWS 會管理升級叢集控制平面。您負責升級資料平面，包括 Fargate Pod 和附加元件。您必須驗證和規劃叢集上執行的工作負載升級，以確保叢集升級後其可用性和操作不會受到影響

就地升級叢集

EKS 支援就地叢集升級策略。這可維護叢集資源，並保持叢集組態的一致性（例如 API 端點、OIDC、ENIs、負載平衡器）。這對叢集使用者來說較不干擾，它會使用叢集中現有的工作負載和資源，而不需要您重新部署工作負載或遷移外部資源（例如 DNS、儲存）。

執行就地叢集升級時，請務必注意一次只能執行一個次要版本升級（例如，從 1.24 到 1.25)。

這表示如果您需要更新多個版本，則需要一系列的循序升級。規劃循序升級更為複雜，且停機時間的風險更高。在這種情況下，請參閱 the section called “評估藍/綠叢集作為就地叢集升級的替代方案”。

依序升級您的控制平面和資料平面

若要升級叢集，您需要採取下列動作：

1. 檢閱 Kubernetes 和 EKS 版本備註。
2. 取得叢集的備份。（選用）
3. 識別和修復已棄用和移除的工作負載中的 API 用量。
4. 如果使用受管節點群組，請確保與控制平面位於相同的 Kubernetes 版本。EKS Fargate Profiles 建立的 EKS 受管節點群組和節點支援 Kubernetes 1.27 版和更新版本的控制平面與資料平面之間的 2 個次要版本扭曲。從 1.28 及更高版本開始，EKS Fargate Profiles 建立的 EKS 受管節點群組和節點支援 3 個次要版本偏斜的 Betweeen 控制平面和資料平面。例如，如果您的 EKS 控制平面版本是 1.28，則可以安全地使用舊有 1.25 的 kubelet 版本。如果您的 EKS 版本是 1.27，您可以使用的最舊 kubelet 版本是 1.25。
5. 使用 AWS 主控台或 cli 升級叢集控制平面。
6. 檢閱附加元件相容性。視需要升級 Kubernetes 附加元件和自訂控制器。
7. 更新 kubectl。
8. 升級叢集資料平面。將節點升級至與升級叢集相同的 Kubernetes 次要版本。

> **Tip**
>
> 如果您的叢集是使用 EKS Auto 模式建立的，則不需要升級叢集資料平面。升級控制平面之後，EKS Auto Mode 會開始逐步更新受管節點，同時遵守所有 Pod 中斷預算。請務必監控這些更新，以確認是否符合您的操作需求。

使用 EKS 文件建立升級檢查清單

EKS Kubernetes 版本文件包含每個版本的詳細變更清單。建立每次升級的檢查清單。

如需特定的 EKS 版本升級指引，請檢閱文件以了解每個版本的顯著變更和考量事項。

- 在標準支援上檢閱 Kubernetes 版本的版本備註
- 在延伸支援上檢閱 Kubernetes 版本的版本備註

使用 Kubernetes API 升級附加元件和元件

升級叢集之前，您應該了解正在使用的 Kubernetes 元件版本。清查叢集元件，並識別直接使用 Kubernetes API 的元件。這包括重要的叢集元件，例如監控和記錄代理程式、叢集自動擴展器、容器儲存驅動程式（例如 EBS CSI、EFS CSI)、輸入控制器，以及直接依賴 Kubernetes API 的任何其他工作負載或附加元件。

> **Tip**
>
> 關鍵叢集元件通常安裝在`*-system`命名空間中

```
kubectl get ns | grep '-system'
```

識別依賴 Kubernetes API 的元件後，請檢查其文件是否有版本相容性和升級需求。例如，請參閱 AWS Load Balancer 控制器文件以取得版本相容性。某些元件可能需要升級或變更組態，才能繼續叢集升級。要檢查的一些關鍵元件包括 CoreDNS、kube-proxy、VPC CNI 和儲存驅動程式。

叢集通常包含許多使用 Kubernetes API 的工作負載，並且是輸入控制器、持續交付系統和監控工具等工作負載功能的必要項目。升級 EKS 叢集時，您還必須升級附加元件和第三方工具，以確保它們相容。

請參閱下列常見附加元件及其相關升級文件的範例：

- Amazon VPC CNI：如需每個叢集版本的建議 Amazon VPC CNI 附加元件版本，請參閱更新 Kubernetes 自我管理附加元件的 Amazon VPC CNI 外掛程式。安裝為 Amazon EKS 附加元件時，一次只能升級一個次要版本。
- kube-proxy：請參閱更新 Kubernetes kube-proxy 自我管理附加元件。
- CoreDNS：請參閱更新 CoreDNS 自我管理附加元件。
- AWS Load Balancer 控制器：AWS Load Balancer 控制器必須與您已部署的 EKS 版本相容。如需詳細資訊，請參閱 安裝指南。
- Amazon Elastic Block Store (Amazon EBS) 容器儲存介面 (CSI) 驅動程式：如需安裝和升級資訊，請參閱將 Amazon EBS CSI 驅動程式作為 Amazon EKS 附加元件管理。
- Amazon Elastic File System (Amazon EFS) 容器儲存介面 (CSI) 驅動程式：如需安裝和升級資訊，請參閱 Amazon EFS CSI 驅動程式。
- Kubernetes Metrics Server：如需詳細資訊，請參閱 GitHub 上的 metrics-server。
- Kubernetes Cluster Autoscaler：若要升級 Kubernetes Cluster Autoscaler 的版本，請在部署中變更映像的版本。Cluster Autoscaler 與 Kubernetes 排程器緊密結合。升級叢集時，您一律需要將其升級。檢閱 GitHub 版本，尋找與您的 Kubernetes 次要版本對應的最新版本地址。
- Karpenter：如需安裝和升級資訊，請參閱 Karpenter 文件。

> **Tip**
>
> 您不需要手動升級 Amazon EKS Auto Mode 的任何功能，包括運算自動擴展、區塊儲存和負載平衡功能。

升級前驗證基本 EKS 需求

AWS 需要帳戶中的特定資源才能完成升級程序。如果這些資源不存在，則無法升級叢集。控制平面升級需要下列資源：

1. 可用的 IP 地址：Amazon EKS 需要您在建立叢集時指定的子網路中最多五個可用的 IP 地址，才能更新叢集。如果沒有，請在執行版本更新之前更新您的叢集組態，以包含新的叢集子網路。
2. EKS IAM 角色：具有必要許可的 帳戶中仍然存在控制平面 IAM 角色。
3. 如果您的叢集已啟用秘密加密，請確定叢集 IAM 角色具有使用 AWS Key Management Service (AWS KMS) 金鑰的許可。

驗證可用的 IP 地址

若要更新叢集，Amazon EKS 最多需要五個來自子網的可用 IP 地址，子網是在您建立叢集時指定的。

若要確認您的子網路有足夠的 IP 地址來升級叢集，您可以執行下列命令：

```
CLUSTER=<cluster name>
aws ec2 describe-subnets --subnet-ids \
  $(aws eks describe-cluster --name ${CLUSTER} \
  --query 'cluster.resourcesVpcConfig.subnetIds' \
  --output text) \
  --query 'Subnets[*].[SubnetId,AvailabilityZone,AvailableIpAddressCount]' \
  --output table

----------------------------------------------------
|                  DescribeSubnets                 |
+---------------------------+--------------+-------+
|  subnet-067fa8ee8476abbd6 |  us-east-1a  |  8184 |
|  subnet-0056f7403b17d2b43 |  us-east-1b  |  8153 |
|  subnet-09586f8fb3addbc8c |  us-east-1a  |  8120 |
|  subnet-047f3d276a22c6bce |  us-east-1b  |  8184 |
+---------------------------+--------------+-------+
```

VPC CNI 指標協助程式可用於為 VPC 指標建立 CloudWatch 儀表板。如果您在建立叢集期間最初指定的子網路中用完 IP 地址，Amazon EKS 建議在開始 Kubernetes 版本升級之前，使用「UpdateClusterConfiguration」 API 更新叢集子網路。請確認將提供給您的新子網路：

- 屬於叢集建立期間選取的相同 AZs 集。
- 屬於叢集建立期間提供的相同 VPC

如果現有 VPC CIDR 區塊中的 IP 地址用完，請考慮關聯其他 CIDR 區塊。AWS 可啟用其他 CIDR 區塊與您現有叢集 VPC 的關聯，有效地擴展您的 IP 地址集區。此擴展可以透過引入其他私有 IP 範圍 (RFC 1918) 或在必要時引入公有 IP 範圍 （非 RFC 1918) 來完成。您必須新增新的 VPC CIDR 區塊，並允許 VPC 重新整理完成，Amazon EKS 才能使用新的 CIDR。之後，您可以根據新設定的 CIDR 區塊，將子網路更新至 VPC。

驗證 EKS IAM 角色

若要驗證 IAM 角色是否可用，且帳戶中具有正確的擔任角色政策，您可以執行下列命令：

```
CLUSTER=<cluster name>
```

```
ROLE_ARN=$(aws eks describe-cluster --name ${CLUSTER} \
  --query 'cluster.roleArn' --output text)
aws iam get-role --role-name ${ROLE_ARN##*/} \
  --query 'Role.AssumeRolePolicyDocument'

{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "eks.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

遷移至 EKS 附加元件

Amazon EKS 會自動為每個叢集安裝附加元件，例如適用於 Kubernetes、 `kube-proxy`和 CoreDNS 的 Amazon VPC CNI 外掛程式。附加元件可以是自我管理的，也可以安裝為 Amazon EKS 附加元件。Amazon EKS 附加元件是使用 EKS API 管理附加元件的替代方式。

您可以使用 Amazon EKS 附加元件，透過單一命令更新版本。例如：

```
aws eks update-addon –cluster-name my-cluster –addon-name vpc-cni –addon-version
 version-number \
--service-account-role-arn arn:aws:iam::111122223333:role/role-name –configuration-
values '{}' –resolve-conflicts PRESERVE
```

檢查是否有任何具有下列項目的 EKS 附加元件：

```
aws eks list-addons --cluster-name <cluster name>
```

> **Warning**
>
> 在控制平面升級期間，EKS 附加元件不會自動升級。您必須啟動 EKS 附加元件更新，然後選取所需的版本。

- 您有責任從所有可用的版本中選取相容的版本。檢閱附加元件版本相容性的指引。
- Amazon EKS 附加元件一次只能升級一個次要版本。

進一步了解哪些元件可作為 EKS 附加元件使用，以及如何開始使用。

了解如何提供自訂組態給 EKS 附加元件。

升級控制平面之前，識別並修復已移除的 API 用量

升級 EKS 控制平面之前，您應該識別已移除 API APIs 使用情況。若要這麼做，我們建議您使用可檢查執行中叢集或靜態轉譯 Kubernetes 資訊清單檔案的工具。

針對靜態資訊清單檔案執行檢查通常更準確。如果針對即時叢集執行，這些工具可能會傳回誤報。

已取代的 Kubernetes API 並不表示 API 已移除。您應該檢查 Kubernetes 棄用政策，以了解 API 移除如何影響您的工作負載。

叢集洞見

Cluster Insights 是一項功能，可提供可能影響將 EKS 叢集升級至較新 Kubernetes 版本之能力的問題調查結果。這些問題清單是由 Amazon EKS 策劃和管理，並提供如何修復問題清單的建議。透過利用 Cluster Insights，您可以將升級到較新 Kubernetes 版本所花費的精力降至最低。

若要檢視 EKS 叢集的洞見，您可以執行 命令：

```
aws eks list-insights --region <region-code> --cluster-name <my-cluster>

{
    "insights": [
        {
            "category": "UPGRADE_READINESS",
            "name": "Deprecated APIs removed in Kubernetes v1.29",
            "insightStatus": {
                "status": "PASSING",
                "reason": "No deprecated API usage detected within the last 30 days."
            },
            "kubernetesVersion": "1.29",
            "lastTransitionTime": 1698774710.0,
            "lastRefreshTime": 1700157422.0,
            "id": "123e4567-e89b-42d3-a456-579642341238",
```

```
            "description": "Checks for usage of deprecated APIs that are scheduled for
 removal in Kubernetes v1.29. Upgrading your cluster before migrating to the updated
 APIs supported by v1.29 could cause application impact."
        }
    ]
}
```

如需所收到洞見的描述性輸出，您可以執行 命令：

```
aws eks describe-insight --region <region-code> --id <insight-id> --cluster-name <my-
cluster>
```

您也可以選擇在 Amazon EKS 主控台中檢視洞見。從叢集清單中選取叢集之後，洞見調查結果會位於 Upgrade Insights 索引標籤下。

如果您使用 找到叢集洞見"status": ERROR，您必須先解決問題，才能執行叢集升級。執行 命令，該aws eks describe-insight命令將分享下列修復建議：

受影響的資源：

```
"resources": [
      {
        "insightStatus": {
          "status": "ERROR"
        },
        "kubernetesResourceUri": "/apis/policy/v1beta1/podsecuritypolicies/null"
      }
]
```

已棄用 APIs：

```
"deprecationDetails": [
      {
        "usage": "/apis/flowcontrol.apiserver.k8s.io/v1beta2/flowschemas",
        "replacedWith": "/apis/flowcontrol.apiserver.k8s.io/v1beta3/flowschemas",
        "stopServingVersion": "1.29",
        "clientStats": [],
        "startServingReplacementVersion": "1.26"
      }
]
```

建議採取的動作：

```
"recommendation": "Update manifests and API clients to use newer Kubernetes APIs if
 applicable before upgrading to Kubernetes v1.26."
```

透過 EKS 主控台或 CLI 利用叢集洞察，有助於加快成功升級 EKS 叢集版本的程序。使用下列資源進一步了解：* 官方 EKS 文件 * Cluster Insights 啟動部落格。

Kube-no-trouble

Kube-no-trouble 是具有命令 的開放原始碼命令列公用程式kubent。當您kubent在沒有任何引數的情況下執行 時，它會使用您目前的 KubeConfig 內容並掃描叢集，並使用要棄用和移除APIs 列印報告。

```
kubent

4:17PM INF >>> Kube No Trouble `kubent` <<<
4:17PM INF version 0.7.0 (git sha d1bb4e5fd6550b533b2013671aa8419d923ee042)
4:17PM INF Initializing collectors and retrieving data
4:17PM INF Target K8s version is 1.24.8-eks-ffeb93d
4:1 INF Retrieved 93 resources from collector name=Cluster
4:17PM INF Retrieved 16 resources from collector name="Helm v3"
4:17PM INF Loaded ruleset name=custom.rego.tmpl
4:17PM INF Loaded ruleset name=deprecated-1-16.rego
4:17PM INF Loaded ruleset name=deprecated-1-22.rego
4:17PM INF Loaded ruleset name=deprecated-1-25.rego
4:17PM INF Loaded ruleset name=deprecated-1-26.rego
4:17PM INF Loaded ruleset name=deprecated-future.rego
_____________________________________________________________________________________
>>> Deprecated APIs removed in 1.25 <<<
-------------------------------------------------------------------------------------
KIND                NAMESPACE     NAME             API_VERSION      REPLACE_WITH
 (SINCE)
PodSecurityPolicy   <undefined>   eks.privileged   policy/v1beta1   <removed> (1.21.0)
```

它也可以用來掃描靜態資訊清單檔案和 helm 套件。建議kubent在持續整合 (CI) 程序中執行 ，以在資訊清單部署之前識別問題。掃描資訊清單也比掃描即時叢集更準確。

Kube-no-trouble 提供範例服務帳戶和角色，具有掃描叢集的適當許可。

Pluto

另一個選項是類似 的 pluto，kubent因為它支援掃描即時叢集、資訊清單檔案、Helm Chart，並具有 GitHub 動作，您可以包含在 CI 程序中。

```
pluto detect-all-in-cluster

NAME             KIND                VERSION          REPLACEMENT   REMOVED
 DEPRECATED   REPL AVAIL
eks.privileged   PodSecurityPolicy   policy/v1beta1                 false     true
    true
```

Resources

若要確認您的叢集在升級之前未使用已棄用 APIs，您應該監控：

- 自 Kubernetes 1.19 版`apiserver_requested_deprecated_apis`起的 指標：

```
kubectl get --raw /metrics | grep apiserver_requested_deprecated_apis

apiserver_requested_deprecated_apis{group="policy",removed_release="1.25",resource="podsecurity
 1
```

- 稽核日誌中的 事件，並將 `k8s.io/deprecated`設定為 `true`：

```
CLUSTER="<cluster_name>"
QUERY_ID=$(aws logs start-query \
 --log-group-name /aws/eks/${CLUSTER}/cluster \
 --start-time $(date -u --date="-30 minutes" "+%s") # or date -v-30M "+%s" on MacOS \
 --end-time $(date "+%s") \
 --query-string 'fields @message | filter `annotations.k8s.io/deprecated`="true"' \
 --query queryId --output text)

echo "Query started (query id: $QUERY_ID), please hold ..." && sleep 5 # give it some
 time to query

aws logs get-query-results --query-id $QUERY_ID
```

如果已棄用的 APIs中，這將輸出行：

```
{
    "results": [
        [
            {
```

```
                "field": "@message",
                "value": "{\"kind\":\"Event\",\"apiVersion\":\"audit.k8s.io/v1\",
\"level\":\"Request\",\"auditID\":\"8f7883c6-b3d5-42d7-967a-1121c6f22f01\",\"stage
\":\"ResponseComplete\",\"requestURI\":\"/apis/policy/v1beta1/podsecuritypolicies?
allowWatchBookmarks=true\\u0026resourceVersion=4131\\u0026timeout=9m19s\
\u0026timeoutSeconds=559\\u0026watch=true\",\"verb\":\"watch\",\"user\":{\"username
\":\"system:apiserver\",\"uid\":\"8aabfade-da52-47da-83b4-46b16cab30fa\",
\"groups\":[\"system:masters\"]},\"sourceIPs\":[\"::1\"],\"userAgent\":\"kube-
apiserver/v1.24.16 (linux/amd64) kubernetes/af930c1\",\"objectRef\":{\"resource
\":\"podsecuritypolicies\",\"apiGroup\":\"policy\",\"apiVersion\":\"v1beta1\"},
\"responseStatus\":{\"metadata\":{},\"code\":200},\"requestReceivedTimestamp\":
\"2023-10-04T12:36:11.849075Z\",\"stageTimestamp\":\"2023-10-04T12:45:30.850483Z\",
\"annotations\":{\"authorization.k8s.io/decision\":\"allow\",\"authorization.k8s.io/
reason\":\"\",\"k8s.io/deprecated\":\"true\",\"k8s.io/removed-release\":\"1.25\"}}"
            },
[...]
```

更新 Kubernetes 工作負載。使用 kubectl-convert 更新資訊清單

確定需要更新哪些工作負載和資訊清單後，您可能需要將資訊清單檔案中的資源類型（例如 PodSecurityPolicies 變更為 PodSecurityStandards)。這將需要更新資源規格，並根據要取代的資源進行額外研究。

如果資源類型保持不變，但需要更新 API 版本，您可以使用 `kubectl-convert`命令自動轉換資訊清單檔案。例如，將較舊的部署轉換為 `apps/v1`。如需詳細資訊，請參閱 Kubernetes 網站上的安裝 kubectl 轉換外掛程式。

```
kubectl-convert -f <file> --output-version <group>/<version>
```

設定 PodDisruptionBudgets 和topologySpreadConstraints，以確保升級資料平面時工作負載的可用性

確保您的工作負載具有適當的 PodDisruptionBudgets 和topologySpreadConstraints，以確保升級資料平面時工作負載的可用性。並非每個工作負載都需要相同的可用性層級，因此您需要驗證工作負載的規模和需求。

確保工作負載分散在多個可用區域中，並在具有拓撲分散的多個主機上，將提高工作負載會自動遷移到新資料平面的可信度，而不會發生事件。

以下是工作負載範例，這些工作負載一律會有 80% 的可用複本，並在區域和主機之間分散複本

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: myapp
spec:
  minAvailable: "80%"
  selector:
    matchLabels:
      app: myapp
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 10
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - image: public.ecr.aws/eks-distro/kubernetes/pause:3.2
        name: myapp
        resources:
          requests:
            cpu: "1"
            memory: 256M
      topologySpreadConstraints:
      - labelSelector:
          matchLabels:
            app: host-zone-spread
        maxSkew: 2
        topologyKey: kubernetes.io/hostname
        whenUnsatisfiable: DoNotSchedule
      - labelSelector:
          matchLabels:
            app: host-zone-spread
        maxSkew: 2
        topologyKey: topology.kubernetes.io/zone
```

```
        whenUnsatisfiable: DoNotSchedule
```

AWS Resilience Hub 已新增 Amazon Elastic Kubernetes Service (Amazon EKS) 做為支援的資源。Resilience Hub 提供單一位置來定義、驗證和追蹤應用程式的彈性，讓您避免因軟體、基礎設施或操作中斷而導致不必要的停機時間。

使用受管節點群組或 Karpenter 簡化資料平面升級

Managed Node Groups 和 Karpenter 都簡化了節點升級，但它們採用不同的方法。

受管節點群組可自動化節點的佈建和生命週期管理。這表示您可以使用單一操作建立、自動更新或終止節點。

在預設組態中，Karpenter 會使用最新的相容 EKS Optimized AMI 自動建立新的節點。隨著 EKS 版本更新 EKS Optimized AMIs或叢集升級，Karpenter 將自動開始使用這些映像。Karpenter 也會實作 Node Expiry 來更新節點。

Karpenter 可設定為使用自訂 AMIs。如果您搭配 Karpenter 使用自訂 AMIs，您必須負責 kubelet 的版本。

確認與現有節點和控制平面的版本相容性

在 Amazon EKS 中繼續進行 Kubernetes 升級之前，請務必確保受管節點群組、自我管理節點和控制平面之間的相容性。相容性取決於您使用的 Kubernetes 版本，並且會根據不同的案例而有所不同。策略：

- Kubernetes v1.28+ — ** 從 Kubernetes 1.28 版及更新版本開始，核心元件有更寬鬆的版本政策。具體而言，Kubernetes API 伺服器與 kubelet 之間的支援偏斜已延伸一個次要版本，從 n-2 到 n-3。例如，如果您的 EKS 控制平面版本是 1.28，則可以安全地使用舊有 1.25 的 kubelet 版本。AWS Fargate、受管節點群組和自我管理節點支援此版本扭曲。基於安全考量，強烈建議將您的 Amazon Machine Image (AMI) 版本up-to-date。較舊的 kubelet 版本可能會因為潛在的常見漏洞與暴露 (CVEs) 而帶來安全風險，這可能會超過使用較舊 kubelet 版本的好處。
- Kubernetes < v1.28 — 如果您使用的版本早於 v1.28，則 API 伺服器與 kubelet 之間的支援扭曲為 n-2。例如，如果您的 EKS 版本是 1.27，您可以使用的最舊 kubelet 版本是 1.25。此版本扭曲適用於 AWS Fargate、受管節點群組和自我管理節點。

啟用 Karpenter 受管節點的節點過期

Karpenter 實作節點升級的一種方式是使用節點過期的概念。這可減少節點升級所需的規劃。當您在佈建器中設定 ttlSecondsUntilExpired 的值時，這會啟用節點過期。在節點以秒為單位達到定義的存留期後，就會安全地耗盡和刪除它們。即使節點正在使用，也可讓您將節點取代為新佈建的升級執行個體。取代節點時，Karpenter 會使用最新的 EKS 最佳化 AMIs。如需詳細資訊，請參閱 Karpenter 網站上的取消佈建。

Karpenter 不會自動將抖動新增至此值。為了防止過度的工作負載中斷，請定義 Pod 中斷預算，如 Kubernetes 文件所示。

如果您在佈建器上設定 ttlSecondsUntilExpired，這適用於與佈建器相關聯的現有節點。

針對 Karpenter 受管節點使用漂移功能

Karpenter 的漂移功能可以自動升級 Karpenter 佈建的節點，以與 EKS 控制平面保持同步。Karpenter 漂移目前需要使用功能閘道啟用。Karpenter 的預設組態會將最新的 EKS 最佳化 AMI 用於與 EKS 叢集控制平面相同的主要和次要版本。

EKS 叢集升級完成後，Karpenter 的漂移功能會偵測 Karpenter 佈建的節點是否針對先前的叢集版本使用 EKS 最佳化 AMIs，並自動封鎖、耗盡和取代這些節點。若要支援移至新節點的 Pod，請遵循 Kubernetes 最佳實務，方法是設定適當的 Pod 資源配額，並使用 Pod 中斷預算 (PDB)。Karpenter 的取消佈建會根據 Pod 資源請求預先鎖定替換節點，並在取消佈建節點時遵守 PDBs。

使用 eksctl 自動化自我管理節點群組的升級

自我管理節點群組是在您的帳戶中部署並連接到 EKS 服務外部叢集的 EC2 執行個體。這些通常由某種形式的自動化工具部署和管理。若要升級自我管理節點群組，您應該參閱工具文件。

例如，eksctl 支援刪除和耗盡自我管理節點。

一些常見的工具包括：

- eksctl
- kOps
- EKS 藍圖

升級之前備份叢集

新版本的 Kubernetes 為您的 Amazon EKS 叢集帶來重大變更。升級叢集之後，就無法降級叢集。

Velero 是一種社群支援的開放原始碼工具，可用於備份現有叢集，並將備份套用至新叢集。

請注意，您只能為 EKS 目前支援的 Kubernetes 版本建立新的叢集。如果目前仍支援叢集執行的版本，且升級失敗，您可以使用原始版本建立新的叢集，並還原資料平面。請注意，包括 IAM 在內的 AWS 資源不會包含在 Velero 的備份中。需要重新建立這些資源。

升級控制平面後重新啟動 Fargate 部署

若要升級 Fargate 資料平面節點，您需要重新部署工作負載。您可以透過使用 `-o wide`選項列出所有 Pod，來識別哪些工作負載正在 Fargate 節點上執行。任何以 開頭的節點名稱`fargate-`都需要在叢集中重新部署。

評估藍/綠叢集作為就地叢集升級的替代方案

有些客戶偏好進行藍/綠升級策略。這可能會有好處，但也包含應考慮的缺點。

優點包括：

- 可以一次變更多個 EKS 版本（例如 1.23 到 1.25)
- 能夠切換回舊叢集
- 建立新的叢集，可使用較新的系統（例如 terraform) 進行管理
- 工作負載可以個別遷移

部分缺點包括：

- 需要更新消費者的 API 端點和 OIDC 變更（例如 kubectl 和 CI/CD)
- 需要在遷移期間平行執行 2 個叢集，這可能會很昂貴並限制區域容量
- 如果工作負載彼此依賴以一起遷移，則需要更多協調
- 負載平衡器和外部 DNS 無法輕鬆跨越多個叢集

雖然可以執行此策略，但它比就地升級更昂貴，並且需要更多時間來進行協調和工作負載遷移。在某些情況下，可能需要它，並且應該仔細規劃。

有了高層級的自動化和 GitOps 等宣告式系統，這可能更容易做到這一點。您將需要針對具狀態工作負載採取額外的預防措施，以便將資料備份並遷移至新的叢集。

如需詳細資訊，請參閱這些部落格文章：

- Kubernetes 叢集升級：藍綠部署策略
- 無狀態 ArgoCD 工作負載的藍/綠或 Canary Amazon EKS 叢集遷移

追蹤 Kubernetes 專案中計劃的主要變更 — 預先考慮

不要只查看下一個版本。在 Kubernetes 發行時檢閱新版本，並識別重大變更。例如，某些應用程式直接使用 Docker API，並在 Kubernetes 中移除對 Docker （ 也稱為 Dockershim) 的容器執行期界面 (CRI) 的支援`1.24`。這種變更需要更多時間來準備。

檢閱您要升級到之版本的所有記錄變更，並記下任何必要的升級步驟。此外，請注意 Amazon EKS 受管叢集特有的任何要求或程序。

- Kubernetes 變更日誌

功能移除的特定指引

移除 1.25 中的 Dockershim - 使用 Detector for Docker Socket (DDS)

適用於 1.25 的 EKS Optimized AMI 不再包含對 Dockershim 的支援。如果您對 Dockershim 有相依性，例如 您要掛載 Docker 通訊端，則必須先移除這些相依性，才能將工作者節點升級至 1.25。

在升級至 1.25 之前，尋找您在 Docker 通訊端上具有相依性的執行個體。建議使用 Detector for Docker Socket (DDS)，這是 kubectl 外掛程式。

移除 1.25 中的 PodSecurityPolicy - 遷移至 Pod 安全標準或policy-as-code解決方案

`PodSecurityPolicy` 在 Kubernetes 1.21 中已棄用，且在 Kubernetes 1.25 中已移除。如果您在叢集中使用 PodSecurityPolicy，則必須先遷移至內建 Kubernetes Pod 安全標準 (PSS) 或policy-as-code解決方案，才能將叢集升級至 1.25 版，以避免工作負載中斷。

AWS 在 EKS 文件中發佈了詳細的常見問答集。

檢閱 Pod 安全標準 (PSS) 和 Pod 安全許可 (PSA) 最佳實務。

檢閱 Kubernetes 網站上的 PodSecurityPolicy 棄用部落格文章。

在 1.23 中棄用樹狀儲存驅動程式 - 遷移至容器儲存介面 (CSI) 驅動程式

Container Storage Interface (CSI) 旨在協助 Kubernetes 取代其現有的樹狀內儲存驅動程式機制。Amazon EKS `1.23` 和更新版本叢集預設啟用 Amazon EBS 容器儲存介面 (CSI) 遷移功能。如果您的 Pod 在版本 `1.22`或更早的叢集上執行，則必須先安裝 Amazon EBS CSI 驅動程式，才能將叢集更新至版本`1.23`，以避免服務中斷。

檢閱 Amazon EBS CSI 遷移常見問答集。

其他資源

ClowdHaus EKS 升級指南

ClowdHaus EKS 升級指南是一種 CLI，可協助升級 Amazon EKS 叢集。它可以分析叢集是否有任何潛在問題，以便在升級之前進行修復。

GoNoGo

GoNoGo 是一種 Alpha 階段工具，用於判斷叢集附加元件的升級可信度。

成本最佳化的最佳實務

成本最佳化是以最低的價格達成您的業務成果。遵循本指南中的文件，即可最佳化 Amazon EKS 工作負載。

一般準則

在雲端中，有許多一般準則可協助您實現微服務的成本最佳化：

- 確保在 Amazon EKS 上執行的工作負載獨立於執行容器的特定基礎設施類型，這將為在最便宜的基礎設施類型上執行它們提供了更大的靈活性。將 Amazon EKS 與 EC2 搭配使用時，由於工作負載的性質，當我們的工作負載需要特定類型的 EC2 執行個體時，可能會有例外狀況，例如需要 GPU 或其他執行個體類型。
- 選取描述最佳的容器執行個體：使用 Amazon CloudWatch Container Insights for Amazon EKS 之類的服務，或 Kubernetes 生態系統中提供的第三方工具，來描述您的生產環境或生產前環境，並監控 CPU 和記憶體等關鍵指標。這將確保我們可以配置正確的資源量，並避免浪費資源。
- 利用 AWS 中提供的不同購買選項，以搭配 EC2 執行 EKS，例如隨需、Spot 和 Savings Plan。

EKS 成本最佳化最佳實務

雲端的成本最佳化有三個一般最佳實務領域：

- 經濟實惠的資源 (Auto Scaling、Down Scaling、政策和購買選項）
- 支出意識（使用 AWS 和第三方工具）
- 隨著時間最佳化（正確調整大小）

如同任何指引，存在權衡。請確定您與您的組織合作，以了解此工作負載的優先順序，以及哪些最佳實務最重要。

本指南的使用方式

本指南適用於負責實作和管理 EKS 叢集及其支援的工作負載的開發團隊。本指南會組織成不同的最佳實務區域，以更輕鬆地使用。每個主題都有建議清單、要使用的工具，以及 EKS 叢集成本最佳化的最佳實務。主題不需要以特定順序讀取。

AWS 服務和 Kubernetes 的主要功能

下列 AWS 服務和功能支援成本最佳化：

- EC2 執行個體類型、S Savings Plan（和預留執行個體）和 Spot 執行個體，價格不同。
- Auto Scaling 搭配 Kubernetes 原生 Auto Scaling 政策。針對可預測的工作負載，考慮 Savings Plan（先前預留的執行個體）。使用 EBS 和 EFS 等受管資料存放區，以取得應用程式資料的彈性和耐久性。
- Billing and Cost Management 主控台儀表板以及 AWS Cost Explorer 提供 AWS 用量的概觀。使用 AWS Organizations 取得精細的帳單詳細資訊。已分享數個第三方工具的詳細資訊。
- Amazon CloudWatch Container Metrics 提供 EKS 叢集資源用量的相關指標。除了 Kubernetes 儀表板之外，Kubernetes 生態系統中還有數種工具可用來減少浪費。

本指南包含一組建議，可用於改善 Amazon EKS 叢集的成本最佳化。

意見回饋

本指南將在 GitHub 上發佈，以便收集更廣泛的 EKS/Kubernetes 社群的直接意見回饋和建議。如果您有最佳實務，認為我們應該包含在指南中，請在 GitHub 儲存庫中提交問題或提交 PR。我們的目的是在服務中新增新功能時或新最佳實務演進時，定期更新指南。

成本最佳化架構

AWS Cloud Economics 是一項紀律，透過採用 Amazon EKS 等現代運算技術，協助客戶提高效率並降低成本。紀律建議遵循名為「雲端財務管理 (CFM) 架構」的方法，其中包含 4 個支柱：

請參閱支柱：衡量和責任

See 支柱是一組基本的活動和技術，定義了如何測量、監控和建立雲端支出的責任。它通常稱為「可觀測性」、「檢測」或「遙測」。「可觀測性」基礎設施的功能和限制決定了可最佳化的項目。清楚了解您的成本是成本最佳化的關鍵第一步，因為您需要知道從何處開始。這種可見性也會引導您需要執行的活動類型，以進一步最佳化您的環境。

以下是我們 See pillar 最佳實務的簡短概觀：

- 為您的工作負載定義和維護標記策略。
  - 使用執行個體標記，標記 EKS 叢集可讓您查看個別叢集成本，並在成本和用量報告中配置這些成本。
- 使用 Kubecost 等技術建立 EKS 用量的報告和監控。
  - 啟用雲端智慧儀表板，透過正確標記資源並使用視覺化，您可以測量和估計成本。
- 將雲端成本分配給應用程式、業務單位 (LoBs) 和收入串流。
- 與業務利益相關者定義、測量和流通效率/價值 KPIs。例如，建立測量每筆交易成本的「單位指標」KPI，例如，乘車共享服務可能具有「每次乘車成本」的 KPI。

如需與此支柱相關的建議技術和活動詳細資訊，請參閱本指南的成本最佳化 - 可觀測性一節。

儲存支柱：成本最佳化

此支柱是以「請參閱」支柱中開發的技術和功能為基礎。下列活動通常屬於此支柱：

- 識別並消除環境中的浪費。
- 符合成本效益的架構師和設計。
- 選擇最佳購買選項，例如隨需執行個體與 Spot 執行個體。
- 隨服務演進而調整：隨著 AWS 服務演進，有效使用這些服務的方式可能會改變。願意適應這些變更。

由於這些活動是可操作的，因此它們高度取決於您環境的特性。問問自己，成本的主要驅動因素是什麼？ 您的不同環境提供哪些商業價值？ 哪些購買選項和基礎設施選擇最適合每個環境，例如執行個體系列類型？

以下是 EKS 叢集最常見的成本驅動因素的優先順序清單：

1. 運算成本：結合多種執行個體系列、購買選項和平衡可擴展性與可用性需要仔細考慮。如需詳細資訊，請參閱本指南的成本最佳化 - 運算一節中的建議。
2. 網路成本：將 3 AZs用於 EKS 叢集可能會增加可用區域間流量成本。如需如何平衡 HA 需求與降低網路流量成本的建議，請參閱本指南的成本最佳化 - 網路一節。
3. 儲存成本：根據 EKS 叢集中工作負載的狀態/無狀態性質，以及如何使用不同的儲存類型，可以將儲存視為工作負載的一部分。如需 EKS 儲存成本的相關考量，請參閱本指南的成本最佳化 - 儲存一節。

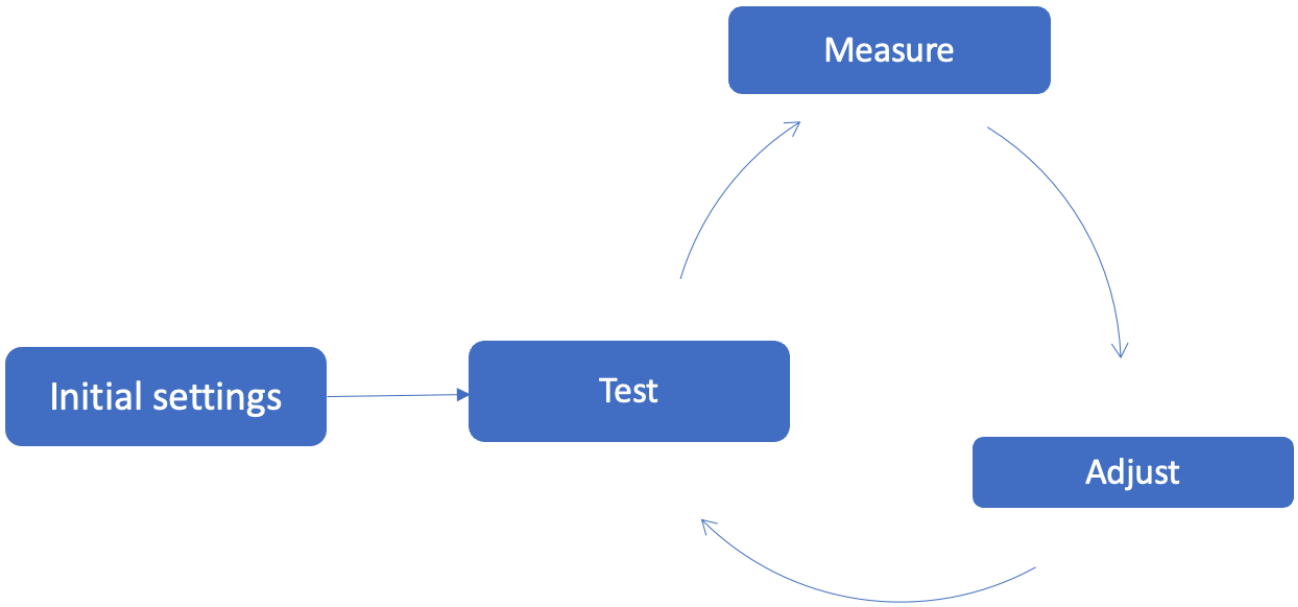
計畫支柱：規劃和預測

一旦實作了 See pillar 中的建議，叢集就會持續進行最佳化。隨著有效獲得操作叢集的經驗，規劃和預測活動可以專注於：

- 動態預算和預測雲端成本。
- 量化 EKS 容器服務交付的商業價值。
- 整合 EKS 叢集成本管理與 IT 財務管理規劃。

執行支柱

成本最佳化是一個持續的過程，涉及漸進式改進的飛輪：



確保這些活動類型的執行贊助對於將 EKS 叢集最佳化整合到組織的「FinOps」工作至關重要。它透過共同了解 EKS 叢集成本、實作 EKS 叢集成本防護機制，以及確保工具、自動化和活動隨著組織需求而發展，來允許利益相關者保持一致。

參考

- AWS 雲端經濟、雲端財務管理

支出意識

支出意識是了解 EKS 叢集中造成支出的人員、位置和原因。準確了解此資料將有助於提高對支出的意識，並重點介紹要修復的領域。

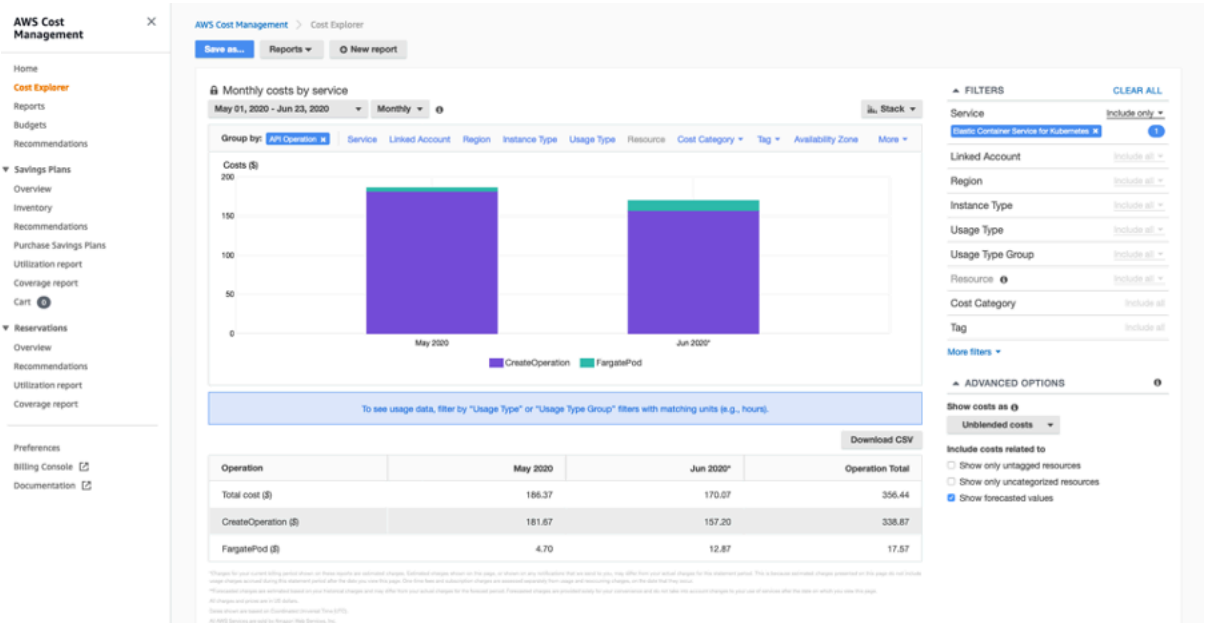
建議

使用 Cost Explorer

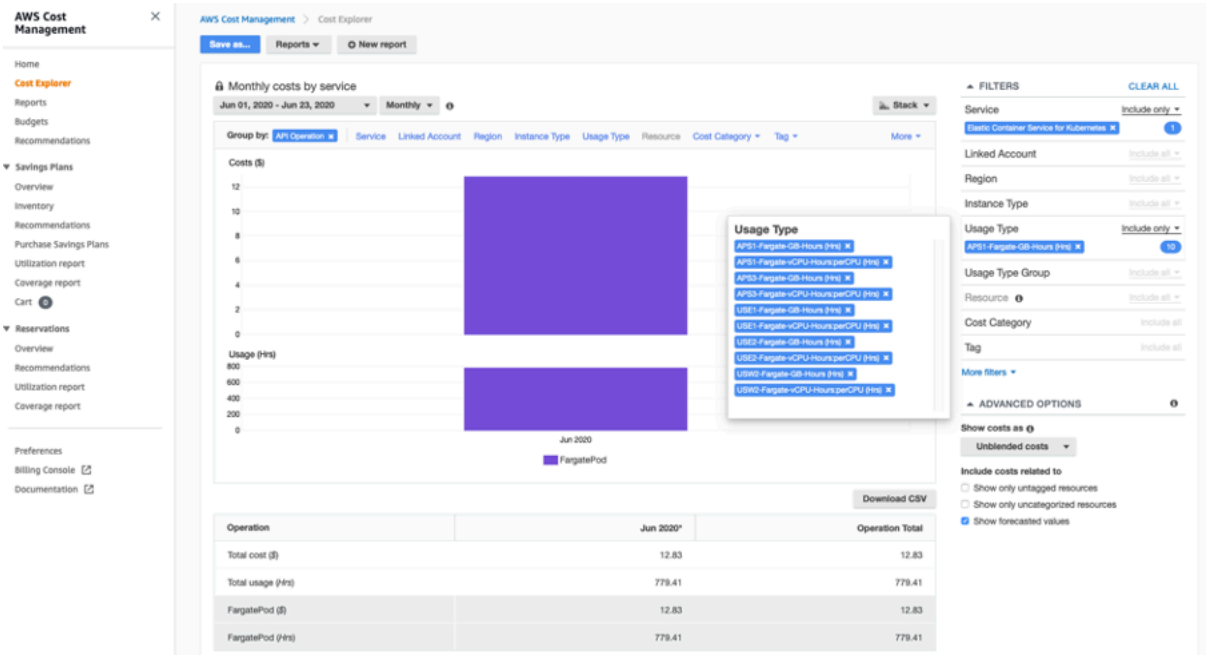
AWS Cost Explorer easy-to-use界面，可讓您視覺化、了解和管理一段時間內的 AWS 成本和用量。您可以使用 Cost Explorer 中提供的篩選條件，在各種層級分析成本和用量資料。

EKS 控制平面和 EKS Fargate 成本

使用篩選條件，我們可以查詢控制平面和 Fargate Pod 的 EKS 成本所產生的成本，如下圖所示：



使用篩選條件，我們可以查詢 EKS 中跨區域 Fargate Pod 產生的彙總成本，其中包括每個 CPU 的 vCPU 小時數和 GB 小時數，如下圖所示：



資源的標記

Amazon EKS 支援將 AWS 標籤新增至您的 Amazon EKS 叢集。這可讓您輕鬆控制對 EKS API 的存取，以管理您的叢集。新增至 EKS 叢集的標籤專屬於 AWS EKS 叢集資源，不會傳播到叢集使用的

其他 AWS 資源，例如 EC2 執行個體或負載平衡器。目前，所有新的和現有的 EKS 叢集都支援透過 AWS API、主控台和 SDKs叢集標記。

AWS Fargate 是一種技術，可為容器提供隨需、大小適中的運算容量。您必須定義至少一個 Fargate 描述檔，指定哪些 Pod 應在啟動時使用 Fargate，才能排程您叢集中 Fargate 上的 Pod。

將標籤新增至 EKS 叢集並列出標籤：

```
$ aws eks tag-resource --resource-arn arn:aws:eks:us-west-2:xxx:cluster/ekscluster1 --
tags team=devops,env=staging,bu=cio,costcenter=1234
$ aws eks list-tags-for-resource --resource-arn arn:aws:eks:us-west-2:xxx:cluster/
ekscluster1
{
    "tags": {
        "bu": "cio",
        "env": "staging",
        "costcenter": "1234",
        "team": "devops"
    }
}
```

在 AWS Cost Explorer 中啟用成本分配標籤後，AWS 會使用成本分配標籤來整理成本分配報告中的資源成本，讓您更輕鬆地分類和追蹤 AWS 成本。

標籤對 Amazon EKS 來說不具有任何語意意義，並會嚴格解譯為字元字串。例如，您可以為 Amazon EKS 叢集定義一組標籤，協助您追蹤每個叢集的擁有者和堆疊層級。

使用 AWS Trusted Advisor

AWS Trusted Advisor 提供一套豐富的最佳實務檢查和五大類別的建議：成本最佳化、安全性、容錯能力、效能和服務限制。

對於成本最佳化，Trusted Advisor 有助於消除未使用的和閒置的資源，並建議承諾預留容量。協助 Amazon EKS 的關鍵動作項目將圍繞低使用率的 EC2 執行個體、未關聯的彈性 IP 地址、閒置負載平衡器、未充分利用的 EBS 磁碟區等。檢查的完整清單提供於 https：//https://aws.amazon.com/premiumsupport/technology/trusted-advisor/best-practice-checklist/。

Trusted Advisor 也為 EC2 執行個體和 Fargate 提供 Savings Plans 和預留執行個體建議，可讓您遞交一致的用量，以換取折扣費率。

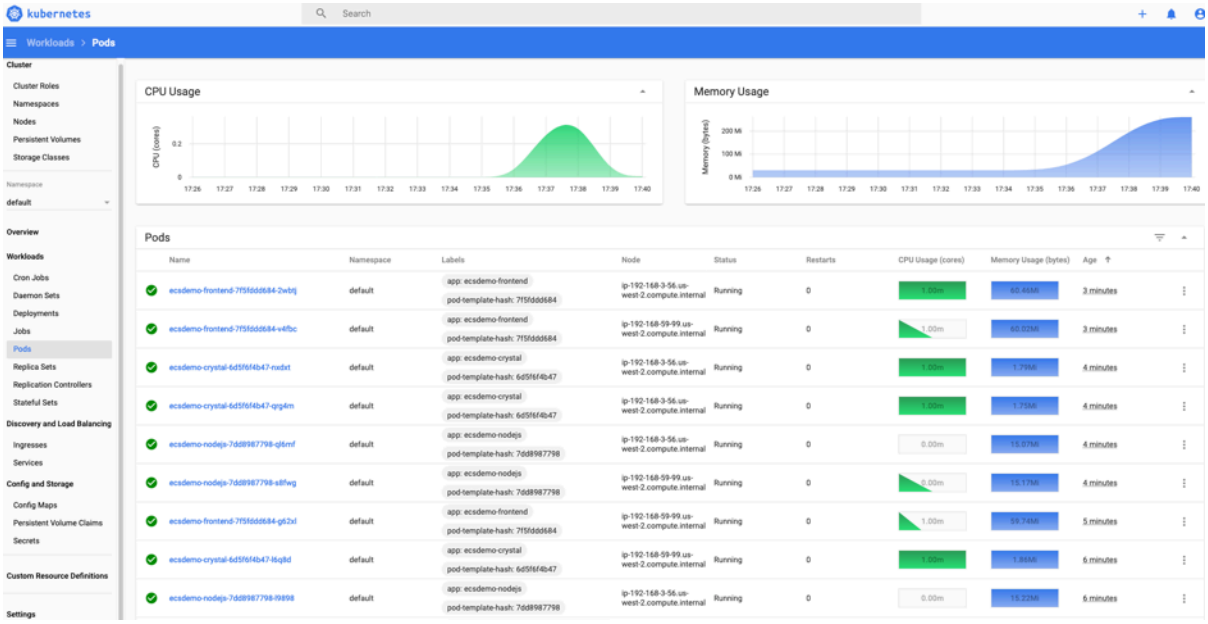
> **Note**
>
> Trusted Advisor 的建議是一般建議，而不是 EKS 特有的建議。

使用 Kubernetes 儀表板

Kubernetes 儀表板

Kubernetes Dashboard 是 Kubernetes 叢集的一般用途 Web 型 UI，可提供 Kubernetes 叢集的相關資訊，包括叢集、節點和 Pod 層級的資源使用量。在 Amazon EKS 叢集上部署 Kubernetes 儀表板的說明請參閱 Amazon EKS 文件。

Dashboard 提供每個節點和 Pod 的資源用量明細，以及有關 Pod、服務、部署和其他 Kubernetes 物件的詳細中繼資料。此合併資訊可讓您了解 Kubernetes 環境。



kubectl top 和描述命令

使用 kubectl top 和 kubectl 來檢視資源用量指標描述命令。kubectl top 會顯示叢集中 Pod 或節點，或特定 Pod 或節點的目前 CPU 和記憶體用量。kubectl describe 命令將提供特定節點或 Pod 的詳細資訊。

```
$ kubectl top pods
$ kubectl top nodes
$ kubectl top pod pod-name --namespace mynamespace --containers
```

使用頂端命令，輸出會顯示節點正在使用的 CPU （以核心為單位） 和記憶體 （以 MiB 為單位） 總量，以及這些數字代表的節點可配置容量百分比。然後，您可以透過新增 --containers 旗標，在 Pod 內向下切入至下一個層級的容器層級。

```
$ kubectl describe node <node>
$ kubectl describe pod <pod>
```

kubectl 描述會傳回每個資源請求或限制所代表的總可用容量百分比。

kubectl top 和 描述、追蹤關鍵資源的使用率和可用性，例如 CPU、記憶體和跨 kubernetes Pod、節點和容器的儲存。此意識有助於了解資源用量，並協助控制成本。

使用 CloudWatch Container Insights

使用 CloudWatch Container Insights 從您的容器化應用程式和微服務收集、彙總和摘要指標和日誌。Container Insights 適用於 EC2 上的 Amazon Elastic Kubernetes Service 和 Amazon EC2 上的 Kubernetes 平台。指標包含 CPU、記憶體、磁碟和網路這類資源的使用率。

文件中會提供洞見的安裝。

CloudWatch 會在叢集、節點、Pod、任務和服務層級建立彙總指標，做為 CloudWatch 指標。

下列查詢顯示節點清單，依平均節點 CPU 使用率排序

```
STATS avg(node_cpu_utilization) as avg_node_cpu_utilization by NodeName
| SORT avg_node_cpu_utilization DESC
```

依容器名稱的 CPU 用量

```
stats pct(container_cpu_usage_total, 50) as CPUPercMedian by kubernetes.container_name
| filter Type="Container"
```

依容器名稱劃分的磁碟用量

```
stats floor(avg(container_filesystem_usage/1024)) as container_filesystem_usage_avg_kb
 by InstanceId, kubernetes.container_name, device
| filter Type="ContainerFS"
| sort container_filesystem_usage_avg_kb desc
```

Container Insights 文件中會提供更多範例查詢

此意識有助於了解資源用量，並協助控制成本。

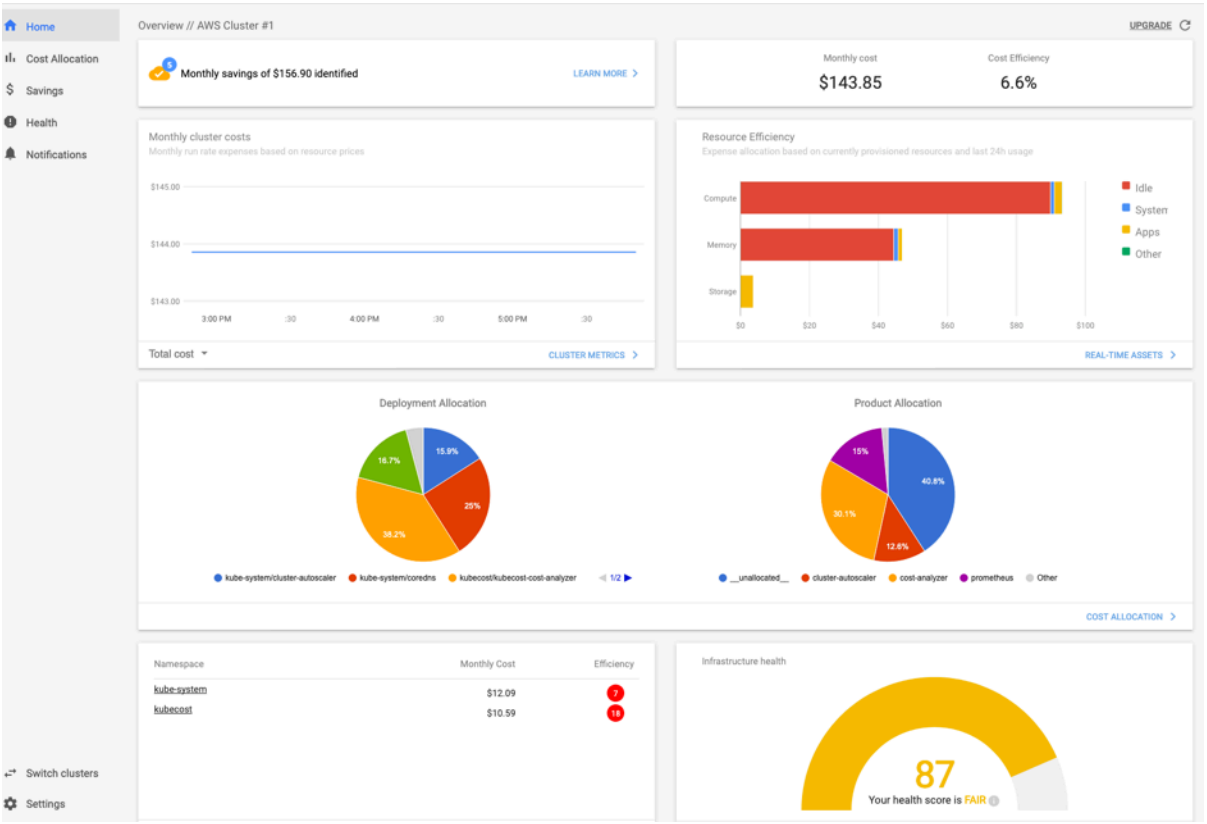
使用 Kubecost 進行支出感知和指導

也可以在 Amazon EKS 上部署 kubecost 等第三方工具，以掌握執行 Kubernetes 叢集的成本。如需使用 Kubecost 追蹤成本的資訊，請參閱此 AWS 部落格

使用 Helm 3 部署 kubecost：

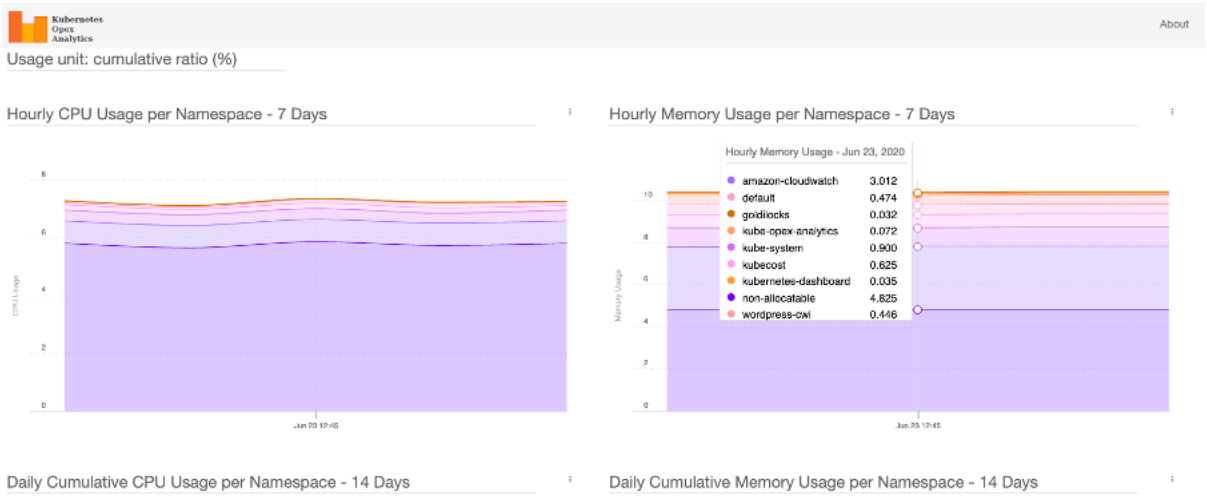
```
$ curl -sSL https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 |
 bash
$ helm version --short
v3.2.1+gfe51cd1
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/c^C
$ kubectl create namespace kubecost
namespace/kubecost created
$ helm repo add kubecost https://kubecost.github.io/cost-analyzer/
"kubecost" has been added to your repositories

$ helm install kubecost kubecost/cost-analyzer --namespace kubecost --set
 kubecostToken="aGRoZEBqc2pzLmNvbQ==xm343yadf98"
NAME: kubecost
LAST DEPLOYED: Mon May 18 08:49:05 2020
NAMESPACE: kubecost
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
--------------------------------------------------Kubecost has been successfully
 installed. When pods are Ready, you can enable port-forwarding with the following
 command:

    kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 9090

Next, navigate to http://localhost:9090 in a web browser.
$ kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 9090

NOTE: If you are using Cloud 9 or have a need to forward it to a different port like
 8080, issue the following command
$ kubectl port-forward --namespace kubecost deployment/kubecost-cost-analyzer 8080:9090
```

Kubecost 儀表板 -



使用 Kubernetes 成本分配和容量規劃分析工具

Kubernetes Opex Analytics 是一種工具，可協助組織追蹤 Kubernetes 叢集所使用的資源，以防止超額付款。為此，它會產生短期 (7 天）、中期 (14 天） 和長期 (12 個月） 用量報告，顯示每個專案隨著時間花費多少資源的相關洞見。



Yotascale

Yotascale 有助於準確分配 Kubernetes 成本。Yotascale Kubernetes 成本分配功能利用實際成本資料，其中包含預留執行個體折扣和 Spot 執行個體定價，而非一般市價估算，以通知 Kubernetes 總成本足跡

如需更多詳細資訊，請參閱其網站。

Alcide Advisor

Alcide 是 AWS 合作夥伴網路 (APN) 進階技術合作夥伴。Alcide Advisor 可協助確保您的 Amazon EKS 叢集、節點和 Pod 組態根據安全最佳實務和內部準則進行調校以執行。Alcide Advisor 是 Kubernetes 稽核和合規的無代理程式服務，其建置目的是在進入生產環境之前強化開發階段，以確保無摩擦且安全的 DevSecOps 流程。

您可以在此部落格文章中找到更多詳細資訊。

其他工具

Kubernetes 垃圾收集

Kubernetes 垃圾收集器的角色是刪除曾經擁有擁有者但不再擁有擁有者的特定物件。

Fargate 計數

Fargatecount 是一種有用的工具，可讓 AWS 客戶使用自訂 CloudWatch 指標，追蹤已部署在特定帳戶特定區域中 Fargate 上的 EKS Pod 總數。這有助於追蹤跨 EKS 叢集執行的所有 Fargate Pod。

Popeye - Kubernetes 叢集清理程式

Popeye - Kubernetes Cluster Sanitizer 是一種公用程式，可掃描即時 Kubernetes 叢集，並報告已部署資源和組態的潛在問題。它會根據部署的內容而非磁碟上的內容來淨化您的叢集。透過掃描您的叢集，它可偵測組態錯誤，並協助您確保最佳實務已就緒

Resources

請參閱下列資源，進一步了解成本最佳化的最佳實務。

文件和部落格

- Amazon EKS 支援標記

工具

- 什麼是 AWS Billing and Cost Management？
- Amazon CloudWatch Container Insights
- 如何使用 Kubecost 追蹤多租戶 Amazon EKS 叢集中的成本
- Kubecost
- Kube Opsview
- Kubernetes Opex Analytics

運算和自動擴展

身為開發人員，您將預估應用程式的資源需求，例如 CPU 和記憶體，但如果您沒有持續調整，它們可能會過期，這可能會增加成本並提高效能和可靠性。持續調整應用程式的資源需求比第一次正確調整它們更為重要。

以下提及的最佳實務將協助您建置和操作具有成本感知的工作負載，以實現業務成果，同時將成本降至最低，並讓您的組織將其投資報酬率最大化。最佳化叢集運算成本的重要高階順序如下：

1. 適當大小的工作負載
2. 減少未使用的容量
3. 最佳化運算容量類型（例如 Spot) 和加速器（例如 GPUs)

適當調整工作負載大小

在大多數 EKS 叢集中，大量成本來自用來執行容器化工作負載的 EC2 執行個體。若未了解工作負載需求，您將無法調整運算資源的大小。因此，您必須使用適當的請求和限制，並視需要調整這些設定。此外，執行個體大小和儲存體選擇等相依性可能會影響工作負載效能，進而對成本和可靠性造成各種意外後果。

請求應與實際使用率相符。如果容器的請求太高，則會有未使用的容量，這是叢集總成本的一大因素。Pod 中的每個容器，例如應用程式和附屬，都應設定自己的請求和限制，以確保彙總 Pod 限制盡可能準確。

利用諸如 Goldilocks、KRR 和 Kubecost 等工具來估計容器的資源請求和限制。根據應用程式的性質、效能/成本需求，以及評估哪些指標最適合擴展、應用程式效能下降的時間點（飽和點），以及如何相應調整請求和限制的複雜性。如需本主題的進一步指引，請參閱應用程式正確調整大小。

我們建議您使用 Horizontal Pod Autoscaler (HPA) 來控制應用程式應執行的複本數量、使用 Vertical Pod Autoscaler (VPA) 來調整應用程式每個複本所需的請求和限制數量，以及使用 Karpenter 或 Cluster Autoscaler 等節點自動擴展器來持續調整叢集中的節點總數。本文件稍後章節會記錄使用 Karpenter 和 Cluster Autoscaler 的成本最佳化技術。

垂直 Pod Autoscaler 可以調整指派給容器的請求和限制，讓工作負載以最佳方式執行。您應該在稽核模式下執行 VPA，這樣它就不會自動進行變更並重新啟動您的 Pod。它會根據觀察到的指標建議變更。對於影響生產工作負載的任何變更，您應該先在非生產環境中檢閱和測試這些變更，因為這些變更可能會影響應用程式的可靠性和效能。

減少耗用量

節省成本的最佳方法是佈建較少的資源。其中一種方法是根據工作負載的目前需求來調整工作負載。您應該從確保您的工作負載定義其需求並動態擴展開始任何成本最佳化工作。這將需要從您的應用程式取得指標和設定組態，例如 `PodDisruptionBudgets`和 Pod 準備階段，以確保您的應用程式可以安全地動態擴展和縮減。請務必考慮限制性 PodDisruptionBudgets 可防止 Cluster Autoscaler 和 Karpenter 縮減節點，因為 Cluster Autoscaler 和 Karpenter 都遵守 PodDisruptionBudgets。PodDisruptionBudget 中的 'minAvailable' 值應一律低於部署中的 Pod 數量，而且您應該在兩個之間保持良好的緩衝，例如，在 6 個 Pod 的部署中，您想要始終至少執行 4 個 Pod，請將 PodDisruptionBidget 中的 'minAvailable' 設定為 4。這將允許 Cluster Autoscaler 和 Karpenter 在節點縮減規模事件期間安全地從未充分利用的節點耗盡和移出 Pod。請參閱 Cluster Autoscaler 常見問答集文件。

Horizontal Pod Autoscaler 是一種靈活的工作負載自動擴展器，可以調整所需的複本數量，以滿足應用程式的效能和可靠性需求。它具有彈性的模型，可根據 CPU、記憶體或自訂指標等各種指標，例如佇列深度、Pod 的連線數量等，來定義何時擴展和縮減。

Kubernetes Metrics Server 可針對 CPU 和記憶體用量等內建指標進行擴展，但如果您想要根據 Amazon CloudWatch 或 SQS 佇列深度等其他指標進行擴展，則應考慮事件驅動型自動擴展專案，例如 KEDA。請參閱此部落格文章，了解如何搭配 CloudWatch 指標使用 KEDA。如果您不確定要根據哪些指標進行監控和擴展，請查看監控重要指標的最佳實務。

減少工作負載耗用量會在叢集中產生多餘的容量，而使用適當的自動擴展組態可讓您自動縮減節點，並減少總支出。建議您不要嘗試手動最佳化運算容量。Kubernetes 排程器和節點自動擴展器旨在為您處理此程序。

減少未使用的容量

確定應用程式的正確大小、減少過多請求後，您可以開始減少佈建的運算容量。如果您已花時間從上述區段正確調整工作負載大小，您應該可以動態執行此操作。AWS 中的 Kubernetes 使用兩個主節點自動擴展器。

Karpenter 和 Cluster Autoscaler

當 Pod 建立或移除且運算需求變更時，Karpenter 和 Kubernetes Cluster Autoscaler 都會擴展叢集中的節點數量。兩者的主要目標是相同的，但 Karpenter 採用不同的節點管理佈建和取消佈建方法，這有助於降低成本和最佳化整個叢集的用量。

隨著叢集的大小和工作負載的增加，預先設定節點群組和執行個體變得更加困難。如同工作負載請求，設定初始基準並根據需要持續調整也很重要。

如果您使用的是 Cluster Autoscaler，它會遵守每個 Auto Scaling 群組 (ASG) 的「最小值」和「最大值」，並且只調整「所需」值。設定基礎 ASG 的這些值時請務必注意，因為 Cluster Autoscaler 無法將 ASG 縮減到超過其「最小」計數。將「所需」計數設定為正常上班時間所需的節點數量，並將「最小」設定為非上班時間所需的節點數量。請參閱 Cluster Autoscaler 常見問答集文件。

Cluster Autoscaler Priority Expander

Kubernetes Cluster Autoscaler 的運作方式是隨著應用程式擴展和縮減，擴展和縮減節點群組，稱為節點群組。如果您不動態擴展工作負載，則 Cluster Autoscaler 不會協助您節省成本。Cluster Autoscaler 需要叢集管理員事先建立節點群組，工作負載才能使用。節點群組需要設定為使用具有相同「設定檔」的執行個體，即大約相同數量的 CPU 和記憶體。

您可以有多個節點群組，而且 Cluster Autoscaler 可以設定為設定優先順序擴展層級，而且每個節點群組可以包含大小不同的節點。節點群組可以有不同的容量類型，而優先順序擴展器可以用來先擴展成本較低的群組。

以下是叢集組態程式碼片段的範例，該程式碼片段使用 `ConfigMap`來排定保留容量的優先順序，然後再使用隨需執行個體。您可以使用相同的技術，將 Graviton 或 Spot 執行個體優先於其他類型。

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: my-cluster
managedNodeGroups:
```

```
  - name: managed-ondemand
    minSize: 1
    maxSize: 7
    instanceType: m5.xlarge
  - name: managed-reserved
    minSize: 2
    maxSize: 10
    instanceType: c5.2xlarge
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-autoscaler-priority-expander
  namespace: kube-system
data:
  priorities: |-
    10:
      - .*ondemand.*
    50:
      - .*reserved.*
```

根據預設，使用節點群組可協助基礎運算資源執行預期項目，例如跨AZs分散節點，但並非所有工作負載都有相同的需求或期望，最好讓應用程式明確宣告其需求。如需 Cluster Autoscaler 的詳細資訊，請參閱最佳實務一節。

取消排程器

Cluster Autoscaler 可以根據需要排程的新 Pod 或節點使用率不足，從叢集新增和移除節點容量。在排程到節點之後，它不會接受 Pod 置放的冪冪檢視。如果您使用的是 Cluster Autoscaler，您也應該查看 Kubernetes 取消排程器，以避免浪費叢集中的容量。

如果您在叢集中有 10 個節點，且每個節點的使用率為 60%，則表示您未使用叢集中佈建容量的 40%。使用 Cluster Autoscaler，您可以將每個節點的使用率閾值設定為 60%，但只有在使用率低於 60% 之後，才會嘗試縮減單一節點。

使用取消排程器，它可以在已排程 Pod 或節點新增至叢集之後查看叢集容量和使用率。它會嘗試將叢集的總容量保持在指定的閾值以上。它也可以根據節點污點或聯結叢集的新節點來移除 Pod，以確保 Pod 在其最佳的運算環境中執行。請注意，取消排程器不會排程取代移出的 Pod，但會依賴預設排程器。

Karpenter 合併

Karpenter 採用「無群組」方法來管理節點。此方法對於不同的工作負載類型更靈活，且叢集管理員需要較少的前期組態。Karpenter 不會視需要預先定義群組和擴展每個群組，而是使用佈建器和節點範本來廣泛定義可建立的 EC2 執行個體類型，以及建立執行個體時的執行個體相關設定。

儲存貯體封裝是將更多工作負載封裝到較少、大小最佳之執行個體上，藉此利用更多執行個體資源的做法。雖然這有助於透過僅佈建工作負載使用的資源來降低運算成本，但它具有權衡。啟動新的工作負載可能需要更長的時間，因為必須將容量新增至叢集，尤其是在大型擴展事件期間。設定儲存貯體封裝時，請考慮成本最佳化、效能和可用性之間的平衡。

Karpenter 可以持續監控和 Binpack，以改善執行個體資源使用率並降低運算成本。Karpenter 也可以為您的工作負載選取更具成本效益的工作者節點。這可以透過在佈建器中將「合併」旗標開啟為 true 來實現（下面的範例程式碼片段）。以下範例顯示啟用整合的範例佈建器。在撰寫本指南時，Karpenter 不會將執行中的 Spot 執行個體取代為價格較低的 Spot 執行個體。如需 Karpenter 整合的詳細資訊，請參閱此部落格。

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
  name: enable-binpacking
spec:
  consolidation:
    enabled: true
```

對於可能無法中斷的工作負載，例如在沒有檢查點的情況下長時間執行的批次任務，請考慮使用 do-not-evict 註釋註釋 Pod。選擇不移出 Pod，表示您告知 Karpenter 不應自願移除包含此 Pod 的節點。不過，如果在節點耗盡時將 Pod do-not-evict 新增至節點，剩餘的 Pod 仍會移出，但該 Pod 會封鎖終止，直到移除為止。在任何一種情況下，都會封鎖節點，以防止在節點上排程其他工作。以下是示範如何設定註釋的範例：

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
  annotations: +
    "karpenter.sh/do-not-evict": "true"
spec:
```

```
  containers:

* name: nginx
image: nginx
ports:
 ** containerPort: 80
```

調整 Cluster Autoscaler 參數以移除未充分利用的節點

節點使用率定義為請求的資源總和除以容量。預設為 `scale-down-utilization-threshold` 50%。此參數可以與 和 搭配使用`scale-down-unneeded-time`，這會決定節點在符合縮減規模資格之前應該不需要多久，預設值為 10 分鐘。在縮減規模的節點上執行的 Pod 將由 kube-scheduler 排程在其他節點上。調整這些設定有助於移除未充分利用的節點，但請務必先測試這些值，以免強制叢集過早縮減規模。

您可以透過確保昂貴的 Pod 受到 Cluster Autoscaler 辨識的標籤保護，來防止縮減規模。若要執行此作業，請確定 Pod 需要昂貴才能移出註釋 `cluster-autoscaler.kubernetes.io/safe-to-evict=false`。以下是設定註釋的範例 yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
  annotations: +
    "cluster-autoscaler.kubernetes.io/safe-to-evict": "false"
spec:
  containers:

* name: nginx
image: nginx
ports:
 ** containerPort: 80
```

使用 Cluster Autoscaler 和 Karpenter 標記節點

AWS 資源標籤可用來組織您的資源，並在詳細的層級追蹤您的 AWS 成本。它們與成本追蹤的 Kubernetes 標籤沒有直接關聯。建議從 Kubernetes 資源標籤開始，並使用 Kubecost 等工具，根據 Pod 上的 Kubernetes 標籤、命名空間等取得基礎設施成本報告。

工作者節點需要有標籤，才能在 AWS Cost Explorer 中顯示帳單資訊。使用 Cluster Autoscaler，使用啟動範本標記受管節點群組中的工作者節點。對於自我管理節點群組，請使用 EC2 自動擴展群組標記您的執行個體。對於 Karpenter 佈建的執行個體，請使用節點範本中的 spec.tags 來標記它們。

多租戶叢集

在由不同團隊共用的叢集上工作時，您可能看不到在相同節點上執行的其他工作負載。雖然資源請求有助於隔離一些「雜訊鄰近」問題，例如 CPU 共用，但它們可能無法隔離所有資源邊界，例如磁碟 I/O 耗盡。不是每個工作負載的消耗性資源都可以隔離或限制。以高於其他工作負載的速率使用共用資源的工作負載，應透過節點污點和容錯進行隔離。這類工作負載的另一項進階技術是 CPU 鎖定，可確保容器的專屬 CPU 而非共用 CPU。

在節點層級隔離工作負載可能更昂貴，但使用預留執行個體、Graviton 處理器或 Spot 可以排程 BestEffort 任務或利用額外的節省。

共用叢集也可能有叢集層級的資源限制，例如 IP 耗盡、Kubernetes 服務限制或 API 擴展請求。您應該檢閱可擴展性最佳實務指南，以確保您的叢集避免這些限制。

您可以在命名空間或 Karpenter 佈建器層級隔離資源。Resource Quotas 提供一種方法來設定對命名空間中資源工作負載可使用數量的限制。這可能是良好的初始護欄，但應該持續評估以確保它不會人為限制工作負載進行擴展。

Karpenter 佈建器可以對叢集中的某些消耗性資源（例如 CPU、GPU) 設定限制，但您需要將租用戶應用程式設定為使用適當的佈建器。這可以防止單一佈建器在叢集中建立太多節點，但應該持續評估，以確保限制不會設定得太低，進而防止工作負載擴展。

排程的自動擴展

您可能需要在週末和下班時間縮減叢集。這對於您想要在不使用時縮減至零的測試和非生產叢集特別相關。叢集倒轉之類的解決方案可以根據 Cron 排程將複本縮減為零。您也可以使用 Karpenter 達成此目標，如下列 AWS 部落格所述。

最佳化運算容量類型

在最佳化叢集中的運算總容量並利用 bin 封裝後，您應該查看您在叢集中佈建的運算類型，以及如何支付這些資源的費用。AWS 具有運算節省計劃，可以降低運算成本，我們將將其分類為下列容量類型：

- Spot
- Savings Plans
- On-Demand

- Fargate

每種容量類型在管理開銷、可用性和長期承諾方面都有不同的權衡，您需要決定哪些適合您的環境。任何環境都不應依賴單一容量類型，而且您可以在單一叢集中混合多個執行類型，以最佳化特定工作負載需求和成本。

Spot

Spot 容量類型會從可用區域中的備用容量佈建 EC2 執行個體。Spot 提供最多 90% 的最大折扣，但這些執行個體可能會在其他地方需要時中斷。此外，佈建新的 Spot 執行個體可能不一定有容量，而且現有的 Spot 執行個體可以透過 2 分鐘的中斷通知回收。如果您的應用程式有很長的啟動或關閉程序，Spot 執行個體可能不是最佳選項。

Spot 運算應使用各種執行個體類型，以減少無法使用 Spot 容量的可能性。需要處理執行個體中斷，才能安全地關閉節點。使用 Karpenter 或受管節點群組的一部分佈建的節點會自動支援執行個體中斷通知。如果您使用的是自我管理節點，則需要分別執行節點終止處理常式，以正常關閉 Spot 執行個體。

您可以在單一叢集中平衡 Spot 執行個體和隨需執行個體。透過 Karpenter，您可以建立加權佈建器，以實現不同容量類型的平衡。使用 Cluster Autoscaler，您可以使用 spot 和隨需或預留執行個體建立混合節點群組。

以下是使用 Karpenter 在隨需執行個體之前排定 Spot 執行個體優先順序的範例。建立佈建器時，您可以指定 Spot、隨需或兩者（如下所示）。當您同時指定 時，如果 Pod 未明確指定是否需要使用 Spot 或隨需，則 Karpenter 會在佈建具有 price-capacity-optimization配置策略 的節點時優先考慮 Spot。

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
  name: spot-prioritized
spec:
  requirements:
    - key: "karpenter.sh/capacity-type"
      operator: In
        values: ["spot", "on-demand"]
```

Savings Plans、預留執行個體和 AWS EDP

您可以使用運算節省計劃來減少運算支出。節省計劃提供 1 或 3 年運算用量承諾的折扣價格。用量可套用至 EKS 叢集中的 EC2 執行個體，但也適用於 Lambda 和 Fargate 等任何運算用量。透過節省計劃，您可以降低成本，並在承諾期間仍然選擇任何 EC2 執行個體類型。

運算節省計劃可將 EC2 成本降低高達 66%，而不需要對您想要使用的執行個體類型、系列或區域做出承諾。使用執行個體時，會自動套用節省成本。

EC2 Instance Savings Plans 提供高達 72% 的運算節省，並承諾在特定區域和 EC2 系列中使用，例如來自 C 系列的執行個體。您可以將用量轉移到區域內的任何 AZ、使用任何世代的執行個體系列，例如 c5 或 c6，以及使用系列中任何大小的執行個體。折扣將自動套用至您帳戶中符合節省計劃條件的任何執行個體。

預留執行個體類似於 EC2 執行個體Savings Plan，但也保證可用區域或區域中的容量，並透過隨需執行個體降低成本，最高可達 72%。計算需要多少預留容量後，您可以選擇要保留多久 (1 年或 3 年）。當您在帳戶中執行這些 EC2 執行個體時，系統會自動套用折扣。

客戶也可以選擇註冊與 AWS 簽訂的 Enterprise 協議。企業協議為客戶提供最符合其需求的專屬協議選項。客戶可以根據 AWS EDP（企業折扣計劃）享有定價折扣。如需 Enterprise Agreements 的詳細資訊，請聯絡您的 AWS 銷售代表。

On-Demand

相較於 Spot，與節省計劃相比，隨需 EC2 執行個體具有無中斷和無長期承諾的可用性優勢。如果您想要降低叢集中的成本，您應該減少隨需 EC2 執行個體的使用量。

最佳化工作負載需求後，您應該能夠計算叢集的最小和最大容量。此數字可能會隨著時間而變更，但很少會下降。考慮將 Savings Plan 用於低於下限的所有項目，並找出不會影響應用程式可用性的容量。任何其他可能無法持續使用或可用性所需的項目都可以隨需使用。

如本節所述，降低用量的最佳方式是盡可能減少資源用量，並充分利用您佈建的資源。使用 Cluster Autoscaler，您可以使用 `scale-down-utilization-threshold`設定移除未充分利用的節點。使用 Karpenter，建議啟用整合。

若要手動識別可與工作負載搭配使用的 EC2 執行個體類型，您應該使用 ec2-instance-selector，它可以顯示每個區域中可用的執行個體，以及與 EKS 相容的執行個體。具有 x86 程序架構、4 Gb 記憶體、2 個 vCPUs 和 us-east-1 區域中可用的執行個體的範例用量。

```
ec2-instance-selector --memory 4 --vcpus 2 --cpu-architecture x86_64 \
  -r us-east-1 --service eks
c5.large
c5a.large
c5ad.large
c5d.large
c6a.large
c6i.large
t2.medium
```

```
t3.medium
t3a.medium
```

對於非生產環境，您可以在未使用的時段自動縮減叢集規模，例如夜間和週末。kubecost 專案叢集倒轉是控制器的範例，可根據設定的排程自動縮減叢集。

Fargate 運算

Fargate 運算是 EKS 叢集的全受管運算選項。它透過在 Kubernetes 叢集中為每個節點排程一個 Pod 來提供 Pod 隔離。它可讓您根據工作負載的 CPU 和 RAM 需求調整運算節點大小，以緊密控制叢集中的工作負載用量。

Fargate 可以將工作負載縮減為 0.5 GB 記憶體的 .25 vCPU，以及 120 GB 記憶體的 16 vCPU。可用的 Pod 大小變化數量有所限制，您需要了解工作負載最適合 Fargate 組態的方式。例如，如果您的工作負載需要 1 個 vCPU 搭配 0.5 GB 的記憶體，最小的 Fargate Pod 將是 1 個 vCPU 搭配 2 GB 的記憶體。

雖然 Fargate 有許多優點，例如沒有 EC2 執行個體或作業系統管理，但它可能需要比傳統 EC2 執行個體更多的運算容量，因為每個部署的 Pod 都會隔離為叢集中的個別節點。這需要更多重複項目，例如 Kubelet、記錄代理程式，以及您通常部署到節點的任何 DaemonSets。Fargate 不支援 DaemonSets，需要將其轉換為 Pod "`sidecars"`，並與應用程式一起執行。

Fargate 無法從 bin 封裝或 CPU 佈建中受益，因為工作負載的界限是無法爆量或在工作負載之間共用的節點。Fargate 將為您節省 EC2 執行個體管理時間，這本身就有成本，但 CPU 和記憶體成本可能比其他 EC2 容量類型更昂貴。Fargate Pod 可以利用運算節省計劃來降低隨需成本。

最佳化運算用量

在運算基礎設施上節省成本的另一種方法是為工作負載使用更有效率的運算。這可以來自效能更高的一般用途運算，例如 Graviton 處理器，比 x86 便宜 20% 且能源效率高 60%，或是 GPUs 和 FPGAs 等工作負載特定加速器。您需要建置可在手臂架構上執行的容器，並使用適合工作負載的加速器設定節點。

EKS 能夠執行具有混合架構（例如 amd64 和 arm64) 的叢集，如果您的容器是針對多個架構編譯的，您可以透過允許佈建器中的兩個架構來利用 Graviton 處理器。不過，若要保持一致的效能，建議您將每個工作負載保留在單一運算架構上，而且只有在沒有額外的容量可用時，才使用不同的架構。

佈建器可以設定多個架構，而工作負載也可以在其工作負載規格中請求特定架構。

```
apiVersion: karpenter.sh/v1
kind: Provisioner
```

```
metadata:
  name: default
spec:
  requirements:
  - key: "kubernetes.io/arch"
    operator: In
    values: ["arm64", "amd64"]
```

使用 Cluster Autoscaler，您將需要為 Graviton 執行個體建立節點群組，並在工作負載上設定節點公差，以利用新的容量。

GPUs 和 FPGAs可以大幅提高工作負載的效能，但工作負載需要最佳化才能使用加速器。許多機器學習和人工智慧的工作負載類型可以使用 GPUs 進行運算，而執行個體可以使用資源請求新增至叢集並掛載到工作負載。

```
spec:
  template:
    spec:
    - containers:
      ...
      resources:
          limits:
            nvidia.com/gpu: "1"
```

有些 GPU 硬體可以跨多個工作負載共用，因此可以佈建和使用單一 GPU。若要了解如何設定工作負載 GPU 共用，請參閱虛擬 GPU 裝置外掛程式以取得詳細資訊。您也可以參考下列部落格：

- 在 Amazon EKS 上，使用 NVIDIA 時間分割和加速的 EC2 執行個體實現 GPU 共用
- 在 Amazon EKS 上，使用 NVIDIA 的多執行個體 GPU (MIG) 最大限度地提高 GPU 使用率：按 GPU 執行更多 Pod 以增強效能

成本最佳化 - 網路

高可用性 (HA) 的架構系統是實現彈性和容錯能力的最佳實務。實際上，這表示將您的工作負載和基礎基礎設施分散到指定 AWS 區域中的多個可用區域 (AZs)。確保您的 Amazon EKS 環境具備這些特性，可增強系統的整體可靠性。除此之外，EKS 環境也可能由各種建構模組（即 VPCs)、元件（即 ELBs) 和整合（即 ECR 和其他容器登錄檔）組成。

高可用性系統與其他使用案例特定元件的組合，在資料傳輸和處理方式上扮演重要角色。這將對資料傳輸和處理所產生的成本產生影響。

以下詳述的實務將協助您設計和最佳化 EKS 環境，以達成不同網域和使用案例的成本效益。

Pod 到 Pod 通訊

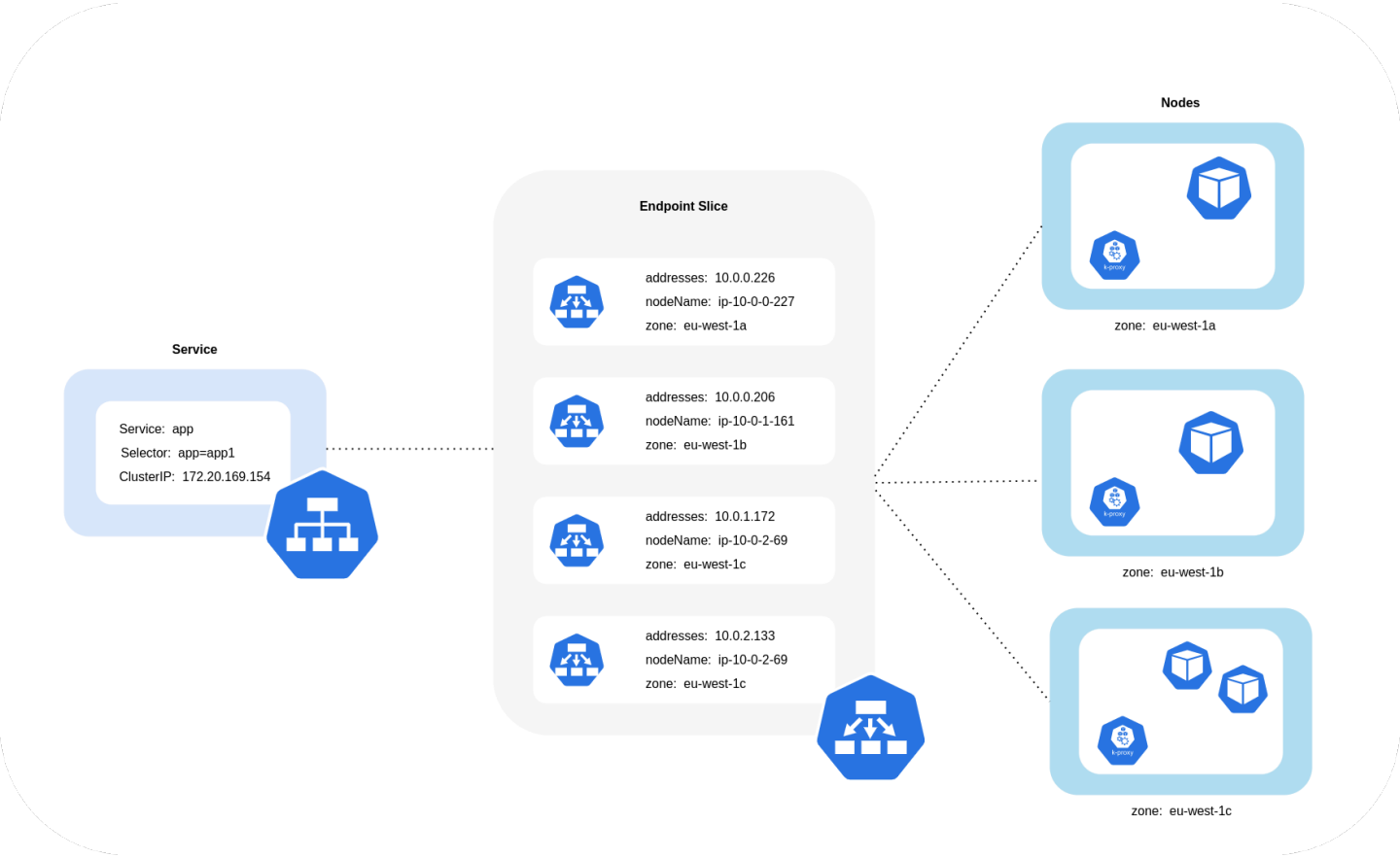
根據您的設定，Pod 之間的網路通訊和資料傳輸可能會對執行 Amazon EKS 工作負載的整體成本產生重大影響。本節將涵蓋不同的概念和方法，以緩解與 Pod 間通訊相關的成本，同時考慮高可用性 (HA) 架構、應用程式效能和彈性。

將流量限制為可用區域

在 的早期，Kubernetes 專案開始開發拓撲感知建構，包括指派給節點的 kubernetes.io/hostname、topology.kubernetes.io/region 和 topology.kubernetes.io/zone 等標籤，以啟用功能，例如跨故障網域和拓撲感知磁碟區佈建器的工作負載分佈。在 Kubernetes 1.17 中畢業後，標籤也被利用來啟用 Pod 到 Pod 通訊的拓撲感知路由功能。

以下是一些策略，說明如何控制 EKS 叢集中 Pod 之間的跨可用區流量，以降低成本並盡可能減少延遲。

如果您想要精細了解叢集中 Pod 之間的跨可用區流量 （例如以位元組為單位傳輸的資料量），請參閱此文章。



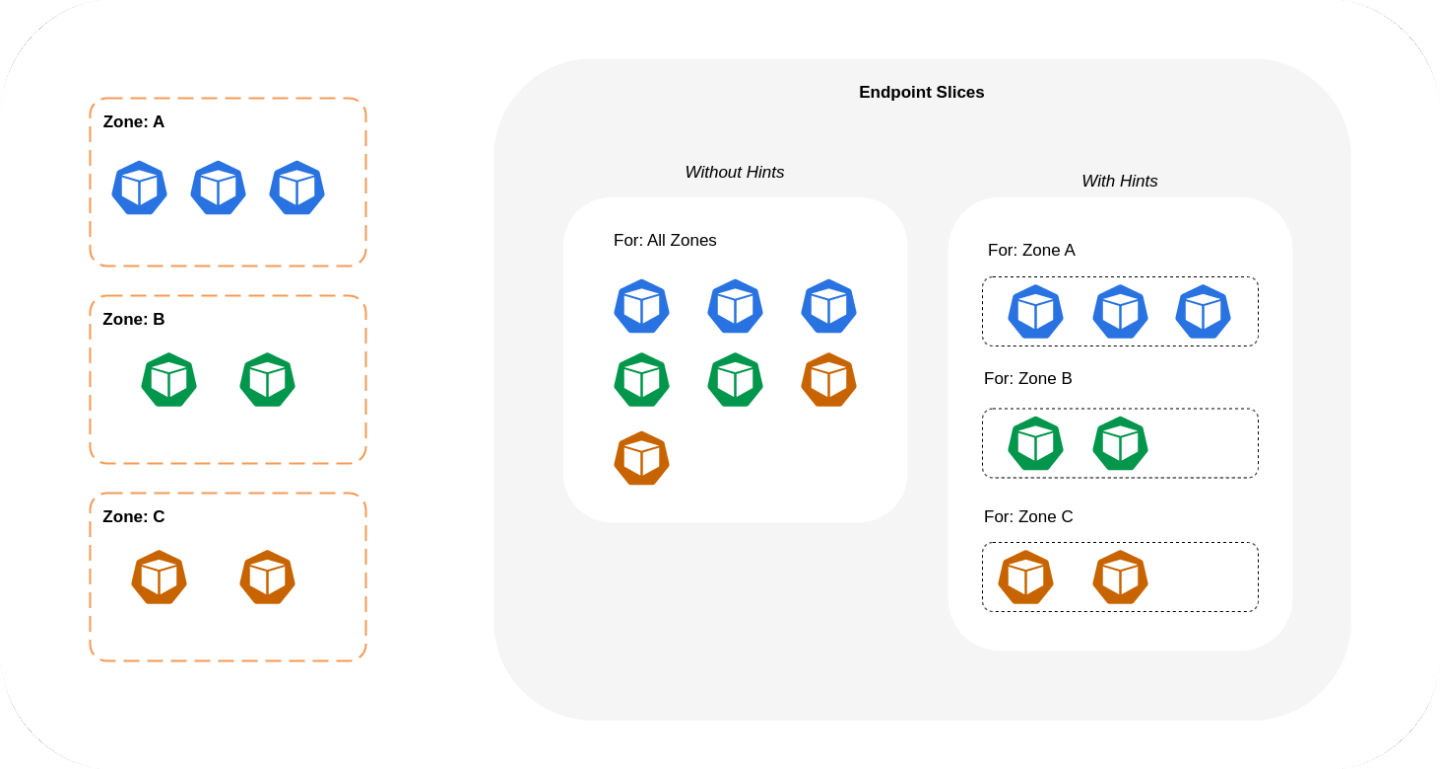
如上圖所示，服務是穩定的網路抽象層，可接收目的地為 Pod 的流量。建立服務時，會建立多個 EndpointSlices。每個 EndpointSlice 都有端點清單，其中包含 Pod 地址的子集，以及它們正在執行的節點，以及任何其他拓撲資訊。使用在每個節點上執行的協助程式集 Amazon VPC CNI 時， kube-proxy 會維護網路規則，以啟用 Pod 通訊和服務探索（替代的 eBPF 型 CNIs 可能不會使用 kube-proxy，但會提供同等行為）。它會履行內部路由的角色，但會根據其從建立的 EndpointSlices 消耗的內容來執行此操作。

在 EKS 上，kube-proxy 主要使用 iptables NAT 規則（或 IPVS、NFTables 作為替代方案）來分佈叢集中所有 Pod 的流量，無論其節點或 AZ 放置為何。此預設分佈可能會導致跨可用區流量路由，這可能會導致敏感應用程式延遲增加，並在大型部署中增加可用區間資料傳輸費用。

使用拓撲感知路由（先前稱為拓撲感知提示）

在 Kubernetes 服務上啟用和實作拓撲感知路由時，EndpointSlice 控制器會將端點按比例配置到叢集分散的不同區域。對於每個端點，EndpointSlice 控制器也會設定區域的提示。提示說明端點應該為哪個區域提供流量。 `kube-proxy`會根據套用的提示，將流量從區域路由到端點。

下圖顯示具有提示的 EndpointSlices 如何組織起來，以便根據其區域來源，`kube-proxy`知道應該前往哪個目的地。如果沒有提示，則不會有此類配置或組織，無論流量來自何處，都會代理到不同的區域目的地。

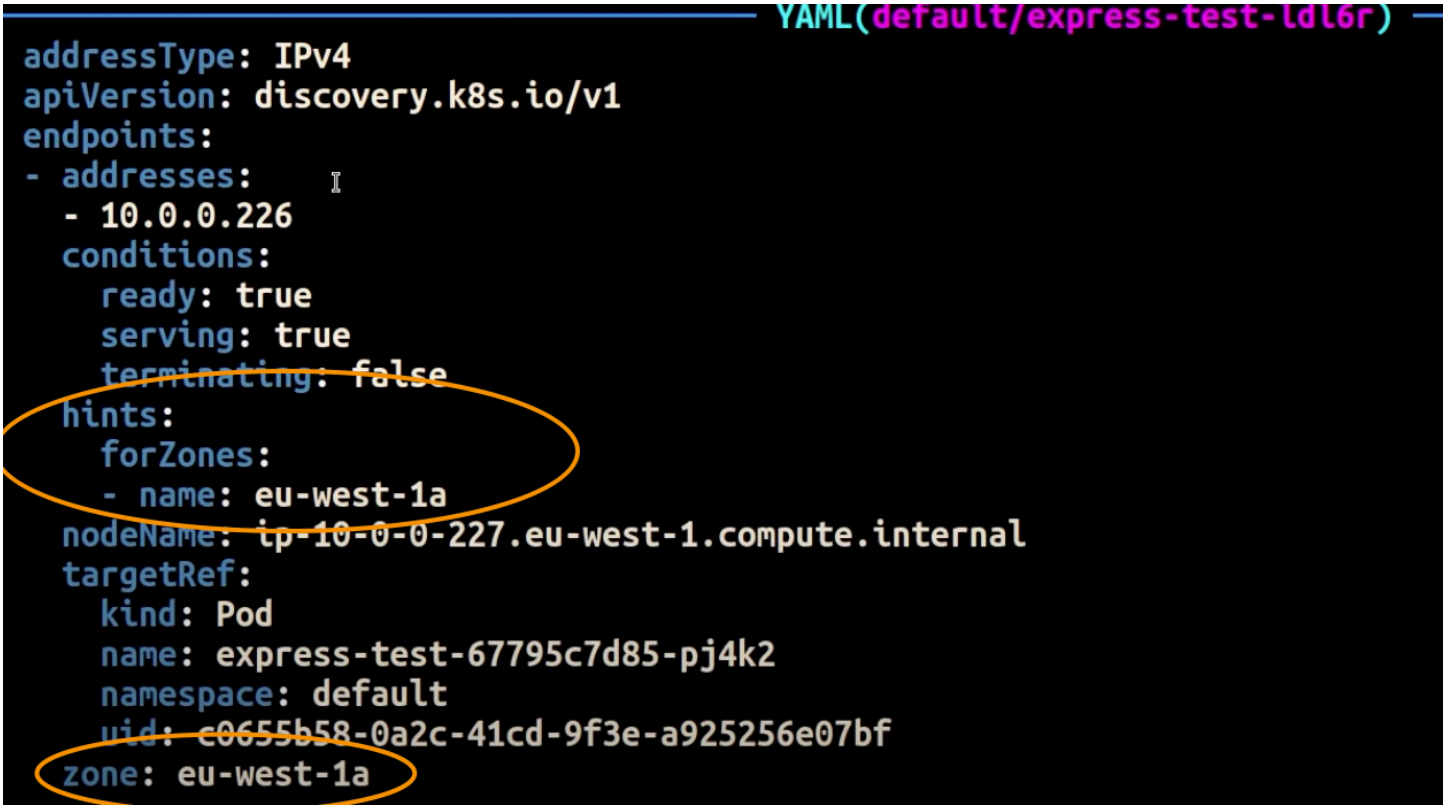


在某些情況下，EndpointSlice 控制器可能會套用不同區域的提示，這表示端點最終可能會為來自不同區域的流量提供服務。這樣做的原因是嘗試和維護不同區域中端點之間的流量均勻分佈。

以下是有關如何啟用服務拓撲感知路由的程式碼片段。

```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
  annotations:
    service.kubernetes.io/topology-mode: Auto
spec:
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
```

以下螢幕擷取畫面顯示 EndpointSlice 控制器的結果，已成功將提示套用至 AZ 中執行之 Pod 複本的端點`eu-west-1a`。



> **Note**
>
> 請務必注意，拓撲感知路由仍在 Beta 版中。此功能在叢集拓撲中平均分佈的工作負載中執行更可預測，因為控制器會按比例分配跨區域的端點，但當區域中的節點資源過於不平衡以避免過載時，可能會略過提示指派。因此，強烈建議將其與排程限制一起使用，以提高應用程式的可用性，例如 Pod 拓撲分散限制。請注意，當容量跨區域波動時，例如使用 Amazon EC2 Spot 執行個體時，可能不會指派提示，因為計算比例分佈時，無法即時偵測到中斷或替換。

使用流量分佈

流量分佈在 Kubernetes 1.30 中推出，並在 1.33 中正式推出，為同一區域流量偏好設定提供了比拓撲感知路由更簡單的替代方案。雖然拓撲感知路由嘗試使用智慧方法來路由流量，以避免端點超載，但會導致無法預測的行為。流量分佈會改為優先考慮可預測性。PreferClose 選項會指示 kube-proxy 建立規則，先根據 EndpointSlice 控制器設定的區域提示，將流量路由到相同區域端點。當沒有相同區域端點可用時，它會回到將流量分散到服務的任何叢集端點。此功能專為接受最佳化鄰近性而非嘗試均勻分佈拓撲感知路由提供之負載的工作負載而設計。

以下是有關如何啟用服務流量分佈的程式碼片段。

```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
spec:
  trafficDistribution: PreferClose
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
```

啟用流量分佈時，會出現常見的挑戰：如果大多數流量來自相同區域，則單一可用區域內的端點可能會過載。此過載可能會造成重大問題：

- 管理多可用區域部署的單一 Horizontal Pod Autoscaler (HPA) 可以透過跨不同AZs擴展 Pod 來回應。不過，此動作無法有效地解決受影響區域中增加的負載。
- 在這種情況下，可能會導致資源效率低下。當類似 Karpenter 的叢集自動擴展器偵測到不同 AZs 的 Pod 橫向擴展時，可能會在不受影響的 AZs 中佈建其他節點，導致不必要的資源配置。

若要克服此挑戰：

- 為每個區域建立單獨的部署，這些區域會有自己的 HPAs 來獨立擴展。
- 利用拓撲分散限制來確保跨叢集的工作負載分佈，這有助於防止高流量區域中的端點過載。

使用自動擴展器：將節點佈建至特定 AZ

我們強烈建議您在跨多個 AZs 的高可用性環境中執行工作負載。這可改善應用程式的可靠性，尤其是在 AZ 發生問題時。如果您願意為了降低網路相關成本而犧牲可靠性，您可以將節點限制為單一可用區域。

若要在相同可用區域中執行所有 Pod，請在相同可用區域中佈建工作者節點，或在相同可用區域中執行的工作者節點上排程 Pod。若要在單一 AZ 中佈建節點，請使用 Cluster Autoscaler (CA) 定義子網路屬於相同 AZ 的節點群組。對於 Karpenter，請使用 `topology.kubernetes.io/zone`並指定您要建立工作者節點的 AZ。例如，以下 Karpenter 佈建器程式碼片段會佈建 us-west-2a AZ 中的節點。

Karpenter

```
apiVersion: karpenter.sh/v1
kind: Provisioner
metadata:
name: single-az
spec:
  requirements:

* key: "topology.kubernetes.io/zone"`
operator: In
values: ["us-west-2a"]
```

Cluster Autoscaler (CA)

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: my-ca-cluster
  region: us-east-1
  version: "1.21"
availabilityZones:

* us-east-1a
managedNodeGroups:
* name: managed-nodes
labels:
  role: managed-nodes
instanceType: t3.medium
minSize: 1
maxSize: 10
desiredCapacity: 1
...
```

使用 Pod 指派和節點親和性

或者，如果您的工作者節點在多個 AZs 中執行，則每個節點都會有標籤 topology.kubernetes.io/zone，其 AZ 值為（例如 us-west-2a 或 us-west-2b)。您可以使用 `nodeSelector`或 `nodeAffinity` 將 Pod 排程到單一 AZ 中的節點。例如，下列資訊清單檔案會在 AZ us-west-2a 中執行的節點內排程 Pod。

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  nodeSelector:
    topology.kubernetes.io/zone: us-west-2a
  containers:

* name: nginx
image: nginx
imagePullPolicy: IfNotPresent
```

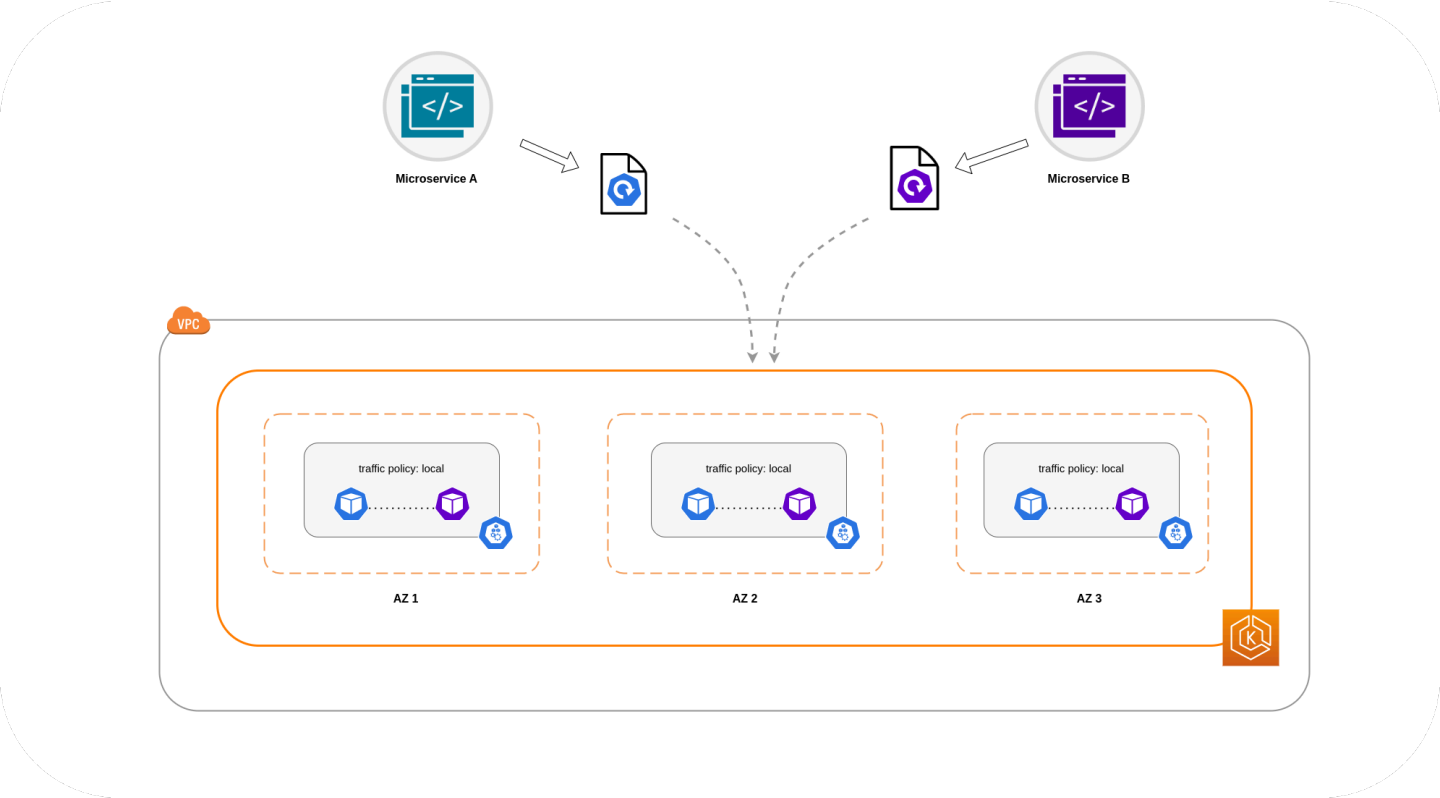
限制節點的流量

在某些情況下，限制區域層級的流量是不夠的。除了降低成本之外，您可能還需要減少某些經常相互通訊的應用程式之間的網路延遲。為了獲得最佳網路效能並降低成本，您需要一種方法來限制特定節點的流量。例如，微服務 A 應一律與節點 1 上的微服務 B 交談，即使在高可用性 (HA) 設定中也是如此。讓節點 1 上的微服務 A 與節點 2 上的微服務 B 交談可能會對此性質的應用程式所需的效能產生負面影響，特別是當節點 2 完全位於單獨的可用區域時。

使用服務內部流量政策

若要將 Pod 網路流量限制為節點，您可以使用服務內部流量政策 。根據預設，傳送至工作負載的服務流量會隨機分散到不同的產生端點。因此，在 HA 架構中，這表示來自 Microservice A 的流量可以在不同 AZs 的任何指定節點上移至 Microservice B 的任何複本。不過，當服務的內部流量政策設為時`Local`，流量將限制為流量來源節點上的端點。此政策指定節點本機端點的獨佔使用。透過隱含，該工作負載的網路流量相關成本將低於分佈的叢集範圍。此外，延遲會較低，讓您的應用程式效能更高。

> **Note**
>
> 請務必注意，此功能無法與 Kubernetes 中的拓撲感知路由結合使用。



以下是有關如何設定服務內部流量政策的程式碼片段。

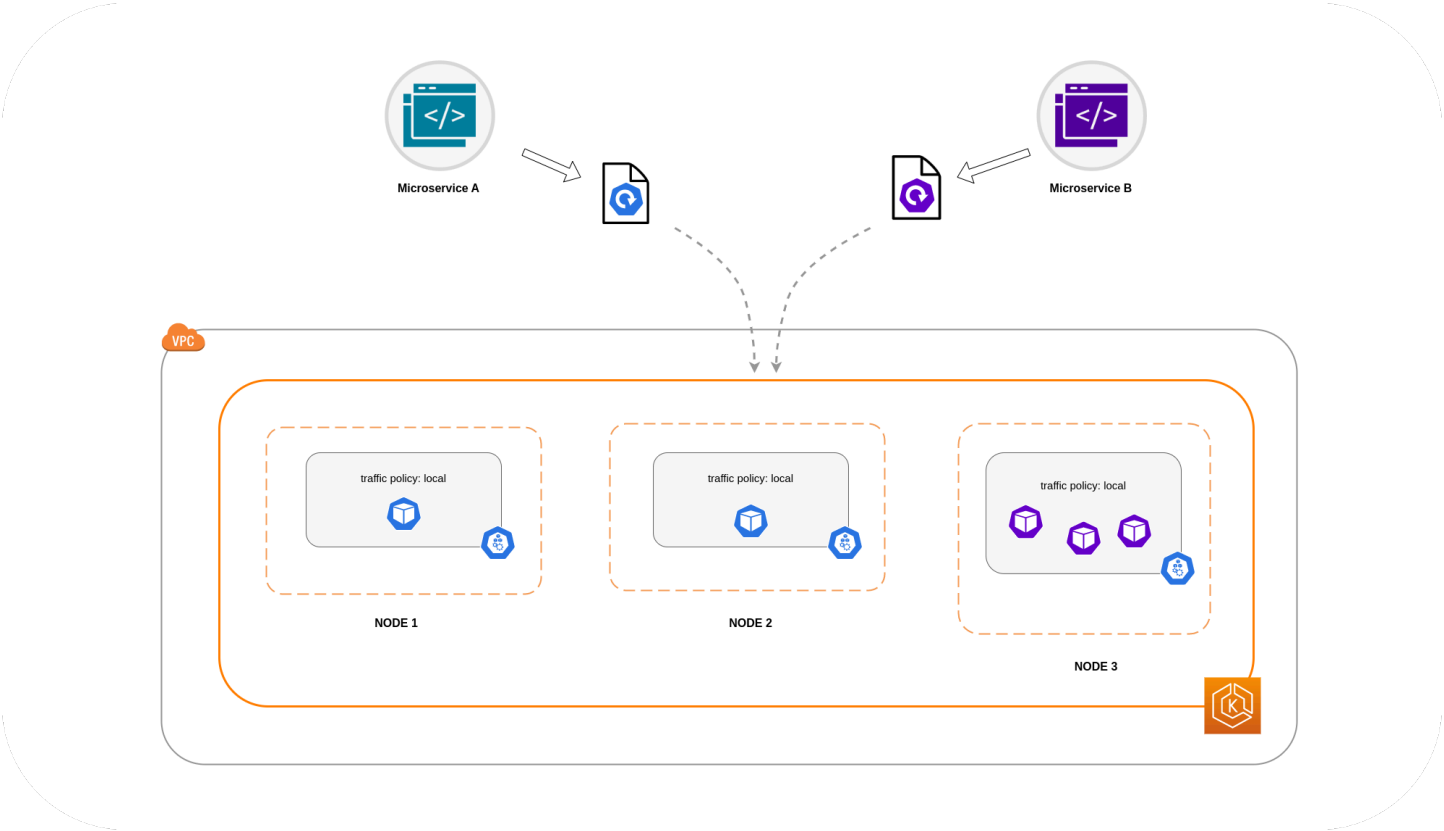
```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: ecommerce
spec:
  selector:
    app: orders
  type: ClusterIP
  ports:

* protocol: TCP
port: 3003
targetPort: 3003
  internalTrafficPolicy: Local
```

為了避免因流量下降而導致應用程式的意外行為，您應該考慮以下方法：

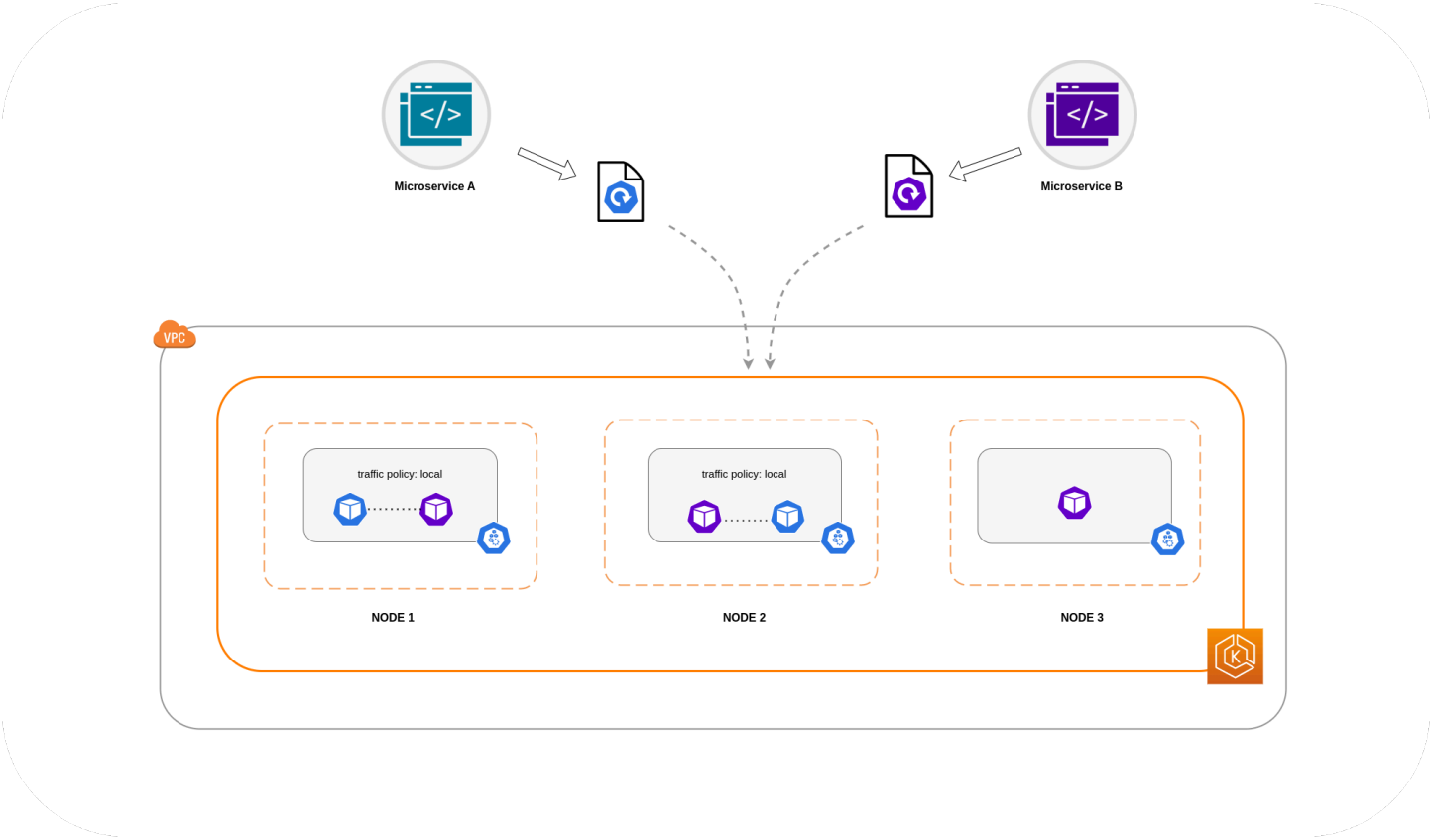
- 為每個通訊 Pod 執行足夠的複本

- 使用拓撲分散限制條件，讓 Pod 的分散相對均勻
- 使用 Pod 親和性規則來共同定位通訊 Pod

在此範例中，您有 2 個 Microservice A 複本和 3 個 Microservice B 複本。如果 Microservice A 的複本分散在節點 1 和 2 之間，且 Microservice B 在節點 3 上有全部 3 個複本，則由於Local內部流量政策，他們將無法通訊。如果沒有可用的節點本機端點，則會捨棄流量。



如果 Microservice B 在節點 1 和 2 上有其 3 個複本中的 2 個，則對等應用程式之間會進行通訊。但是，您仍然會有 Microservice B 的隔離複本，而沒有任何對等複本可與之通訊。



> **Note**
>
> 在某些情況下，如上圖中描述的隔離複本，如果仍有用途（例如提供來自外部傳入流量的請求），則可能不是需要關注的原因。

將服務內部流量政策與拓撲分散限制搭配使用

搭配使用內部流量政策與拓撲分散限制，有助於確保您擁有適當數量的複本，以在不同節點上通訊微服務。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: express-test
spec:
  replicas: 6
  selector:
    matchLabels:
      app: express-test
```

```
  template:
    metadata:
      labels:
        app: express-test
        tier: backend
    spec:
      topologySpreadConstraints:
      - maxSkew: 1
        topologyKey: "topology.kubernetes.io/zone"
        whenUnsatisfiable: ScheduleAnyway
        labelSelector:
          matchLabels:
            app: express-test
```

搭配 Pod 親和性規則使用服務內部流量政策

另一種方法是在使用服務內部流量政策時，使用 Pod 親和性規則。使用 Pod 親和性，您可以影響排程器來聯合定位某些 Pod，因為它們經常通訊。透過在特定 Pod 上套用嚴格的排程限制 (`requiredDuringSchedulingIgnoredDuringExecution`)，當排程器將 Pod 放置在節點上時，這將為您提供更好的 Pod 共置結果。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: graphql
  namespace: ecommerce
  labels:
    app.kubernetes.io/version: "0.1.6"
    ...
    spec:
      serviceAccountName: graphql-service-account
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - orders
            topologyKey: "kubernetes.io/hostname"
```

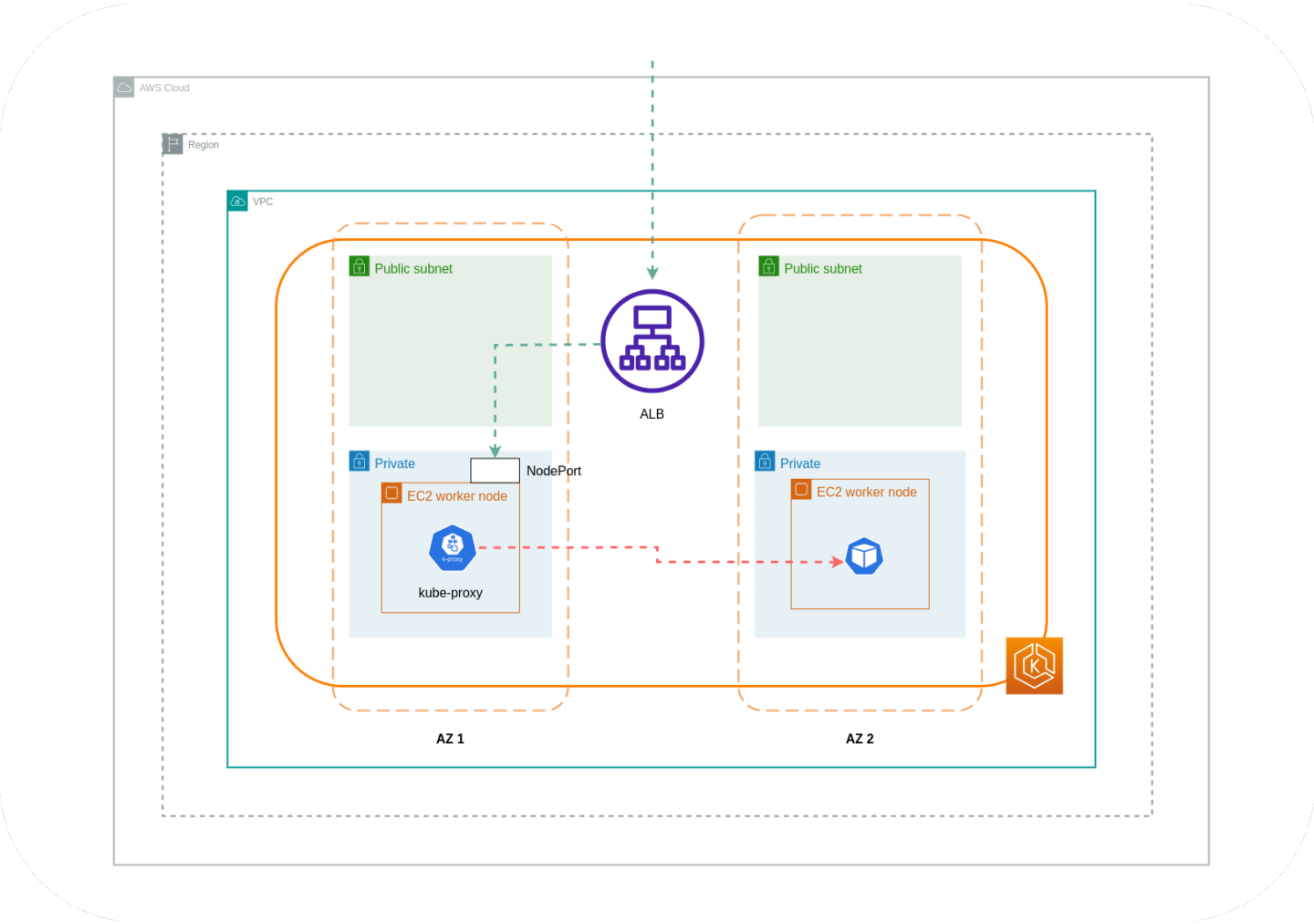
Load Balancer與 Pod 通訊

EKS 工作負載通常由負載平衡器前置，將流量分配到 EKS 叢集中的相關 Pod。您的架構可能包含內部和/或面向外部的負載平衡器。根據您的架構和網路流量組態，負載平衡器和 Pod 之間的通訊可能會產生大量的資料傳輸費用。

您可以使用 AWS Load Balancer 控制器自動管理 ELB 資源 (ALB 和 NLB) 的建立。您在此類設定中產生的資料傳輸費用將取決於網路流量採取的路徑。AWS Load Balancer 控制器支援兩種網路流量模式：執行個體模式和 IP 模式。

使用執行個體模式時，會在 EKS 叢集中的每個節點上開啟 NodePort。然後，負載平衡器會跨節點平均代理流量。如果節點上執行目的地 Pod，則不會產生資料傳輸費用。不過，如果目的地 Pod 位於另一個節點，且位於與接收流量的 NodePort 不同的 AZ 中，則會有從 kube-proxy 到目的地 Pod 的額外網路跳轉。在這種情況下，會產生跨可用區域資料傳輸費用。由於跨節點的流量均勻分佈，因此很可能會產生與跨區域網路流量跳躍相關的額外資料傳輸費用，從 kube-proxies 跳至相關目的地 Pod。

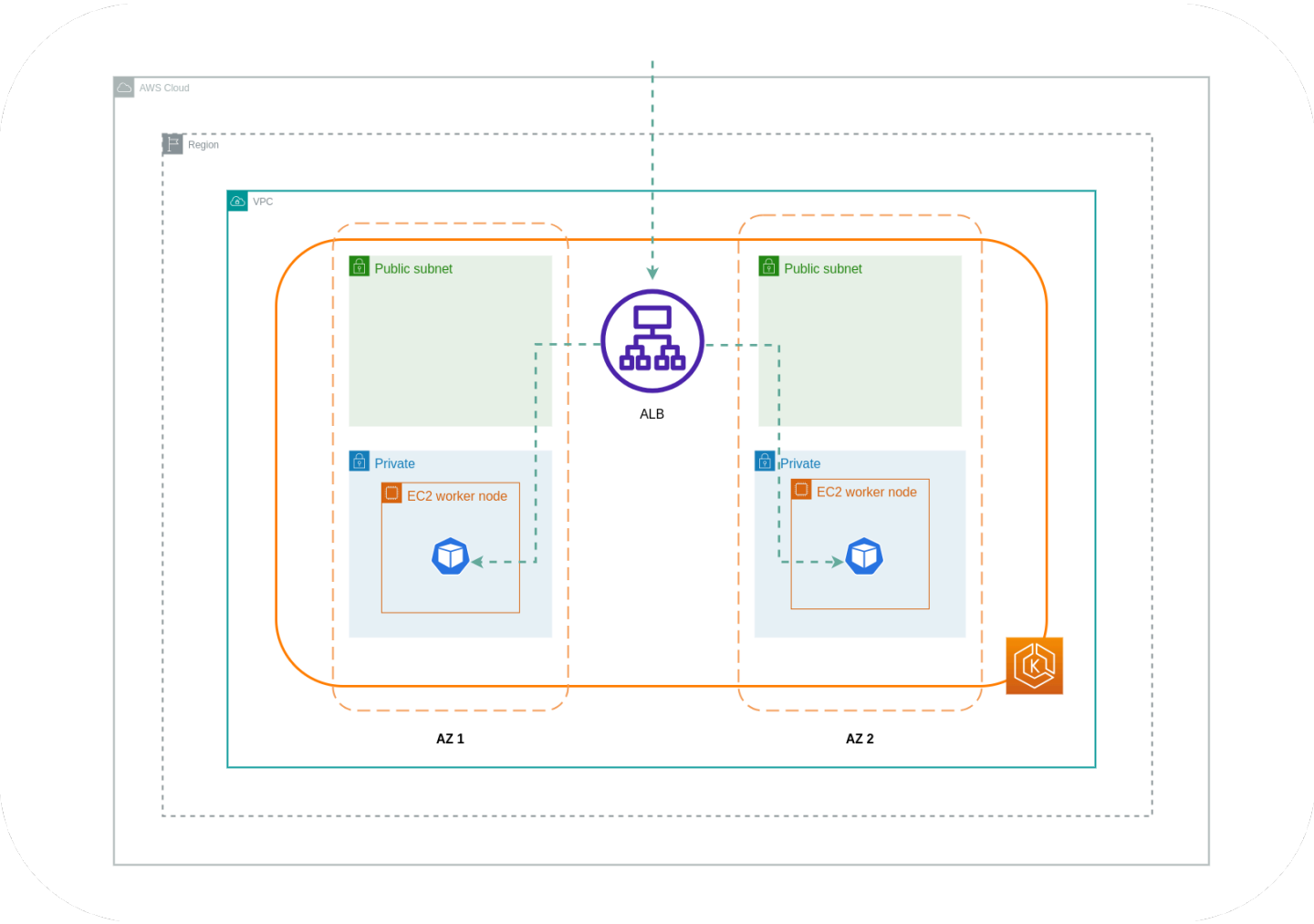
下圖說明從負載平衡器流向 NodePort，然後從 `kube-proxy`流向不同 AZ 中不同節點上目的地 Pod 之流量的網路路徑。這是執行個體模式設定的範例。



使用 ip 模式時，網路流量會從負載平衡器直接代理至目的地 Pod。因此，此方法不會涉及任何資料傳輸費用。

> **Note**
>
> 建議您將負載平衡器設定為 IP 流量模式，以減少資料傳輸費用。對於此設定，確保您的負載平衡器部署在 VPC 中的所有子網路中也很重要。

下圖說明網路 IP 模式下從負載平衡器流向 Pod 之流量的網路路徑。



從容器登錄檔傳輸資料

Amazon ECR

將資料傳輸到 Amazon ECR 私有登錄檔是免費的。區域內資料傳輸不會產生任何費用，但傳輸至網際網路和跨區域的資料將按傳輸兩端的網際網路資料傳輸費率收費。

您應該使用 ECRs內建映像複寫功能，將相關的容器映像複寫到與工作負載相同的區域。如此一來，複寫就會收取一次費用，而且所有相同的區域（區域內）映像提取都是免費的。

您可以使用介面 VPC 端點連線至區域內 ECR 儲存庫，進一步降低從 ECR 提取映像（資料傳輸）的相關資料傳輸成本。連線至 ECR 公有 AWS 端點（透過 NAT Gateway 和網際網路閘道）的替代方法會產生更高的資料處理和傳輸成本。下一節將詳細說明降低工作負載與 AWS Services 之間的資料傳輸成本。

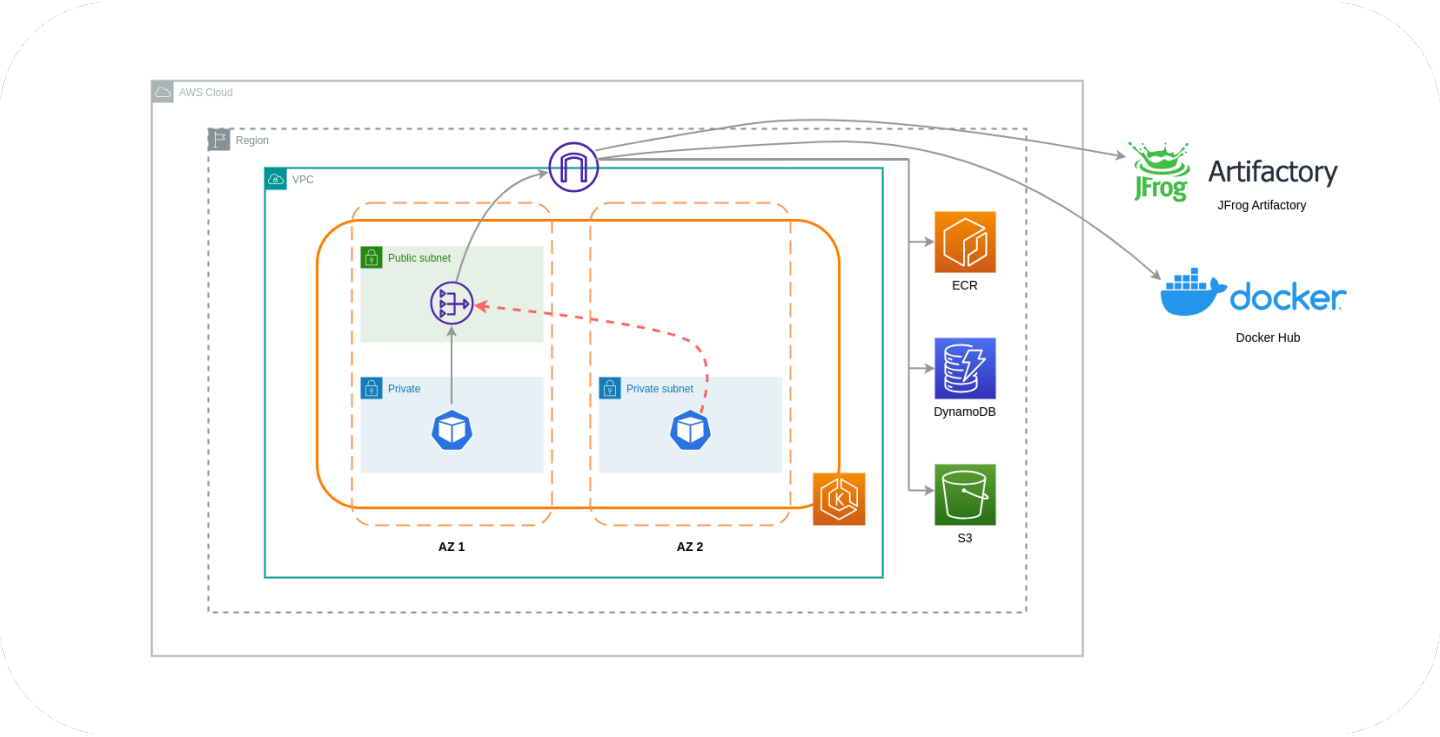
如果您使用特別是大型映像執行工作負載，則可以使用預先快取的容器映像建置自己的自訂 Amazon Machine Image (AMIs)。這可以減少從容器登錄檔到 EKS 工作者節點的初始映像提取時間和潛在的資料傳輸成本。

資料傳輸至網際網路和 AWS 服務

這是透過網際網路將 Kubernetes 工作負載與其他 AWS 服務或第三方工具和平台整合的常見做法。用於往返相關目的地流量的基礎網路基礎設施可能會影響資料傳輸過程中產生的成本。

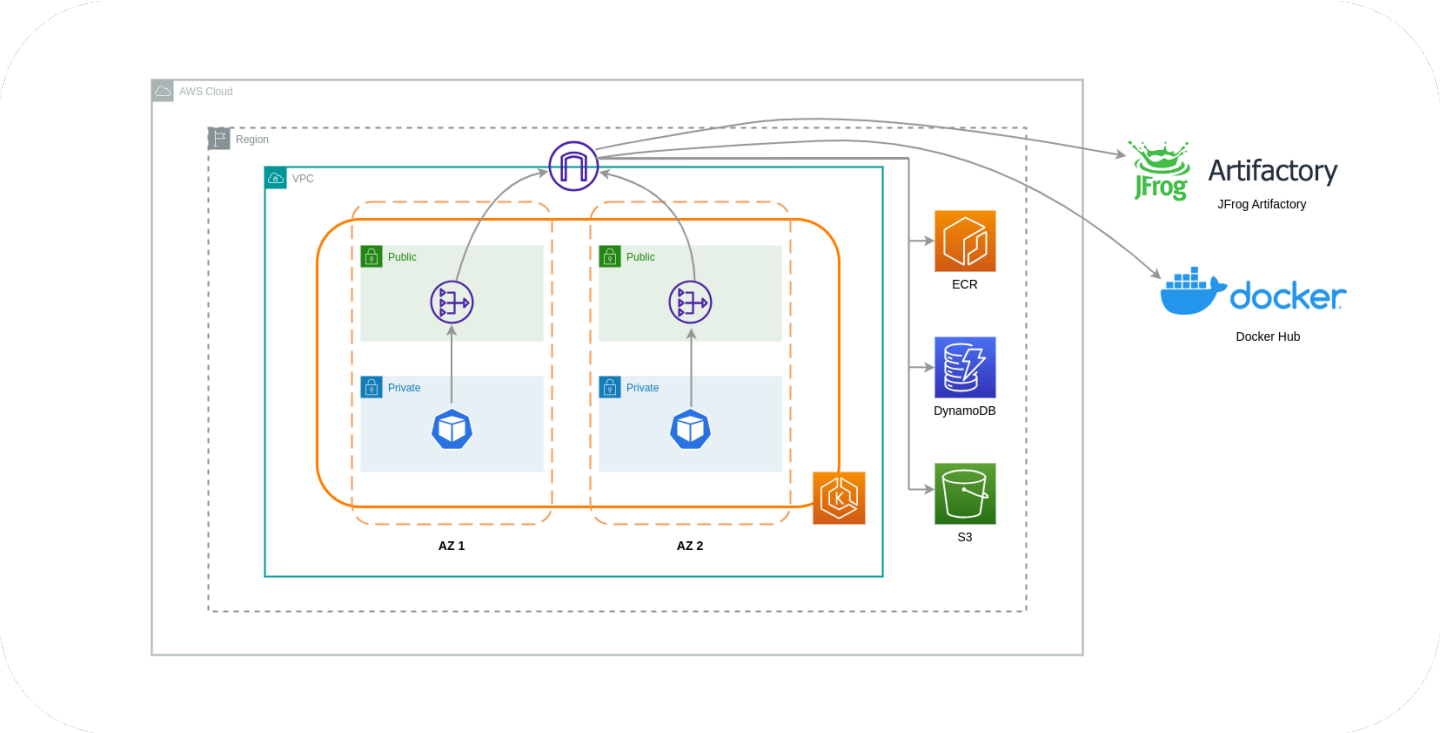
使用 NAT 閘道

NAT Gateways 是執行網路位址轉譯 (NAT) 的網路元件。下圖說明 EKS 叢集中與其他 AWS 服務 (Amazon ECR、DynamoDB 和 S3) 和第三方平台通訊的 Pod。在此範例中，Pod 會在個別 AZs 的私有子網路中執行。若要從網際網路傳送和接收流量，NAT Gateway 會部署到一個可用區域的公有子網路，允許具有私有 IP 地址的任何資源共用單一公有 IP 地址以存取網際網路。此 NAT Gateway 會接著與網際網路閘道元件通訊，允許封包傳送到其最終目的地。



針對這類使用案例使用 NAT Gateway 時，您可以在每個 AZ 中部署 NAT Gateway，將資料傳輸成本降至最低。如此一來，路由至網際網路的流量將通過相同可用區域中的 NAT Gateway，避免跨可用區域資料傳輸。不過，即使您將節省異地同步備份資料傳輸的成本，此設定的隱含性是您架構中其他 NAT Gateway 的成本。

此建議的方法如下圖所示。



使用 VPC 端點

若要進一步降低此類架構的成本，您應該使用 VPC 端點在工作負載和 AWS 服務之間建立連線。VPC 端點可讓您從 VPC 內存取 AWS 服務，無需資料/網路封包周遊網際網路。所有流量都是內部的，並保留在 AWS 網路中。VPC 端點有兩種類型：界面 VPC 端點 (許多 AWS 服務支援) 和閘道 VPC 端點（僅 S3 和 DynamoDB 支援）。

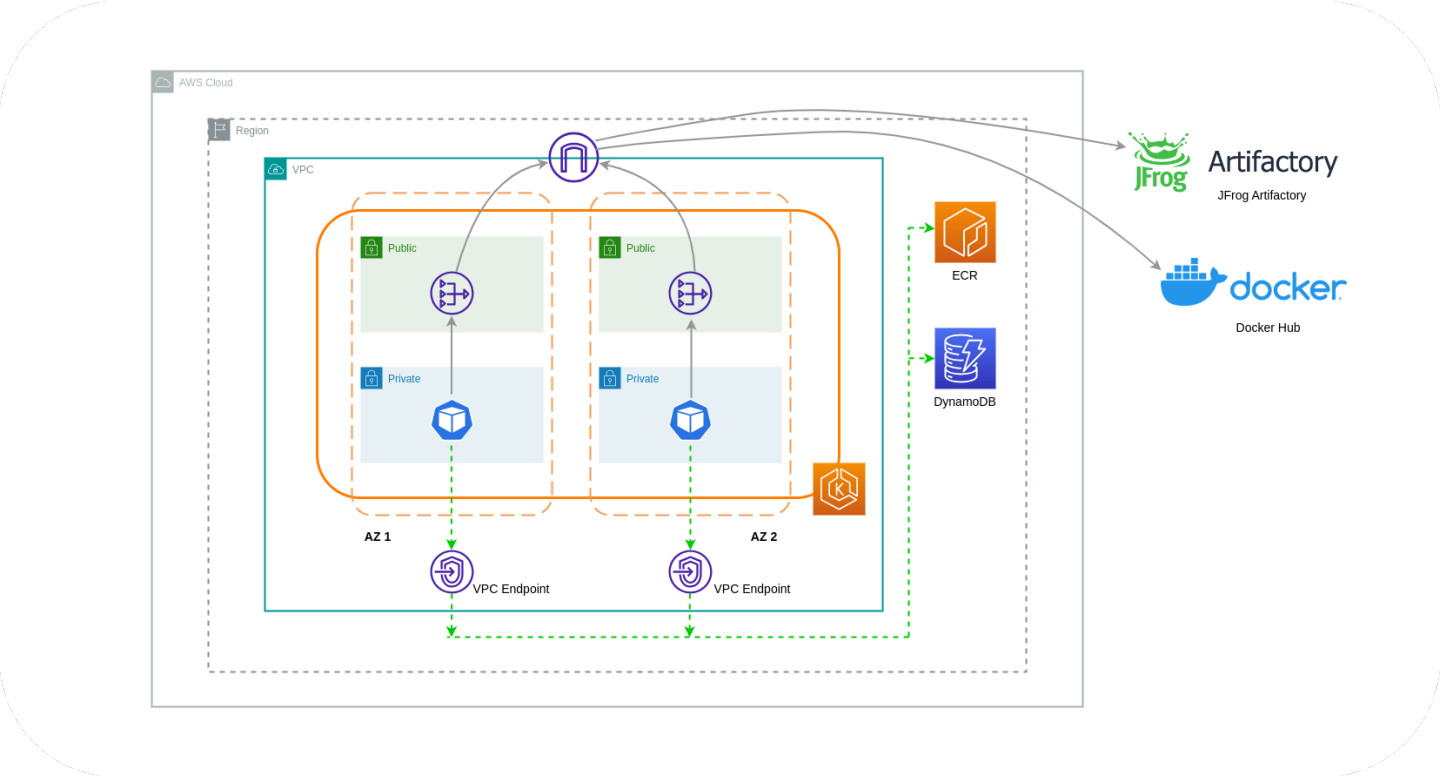
閘道 VPC 端點

閘道 VPC 端點沒有相關的每小時或資料傳輸成本。使用閘道 VPC 端點時，請務必注意它們無法跨 VPC 邊界延伸。它們無法用於 VPC 對等互連、VPN 聯網或透過 Direct Connect 使用。

介面 VPC 端點

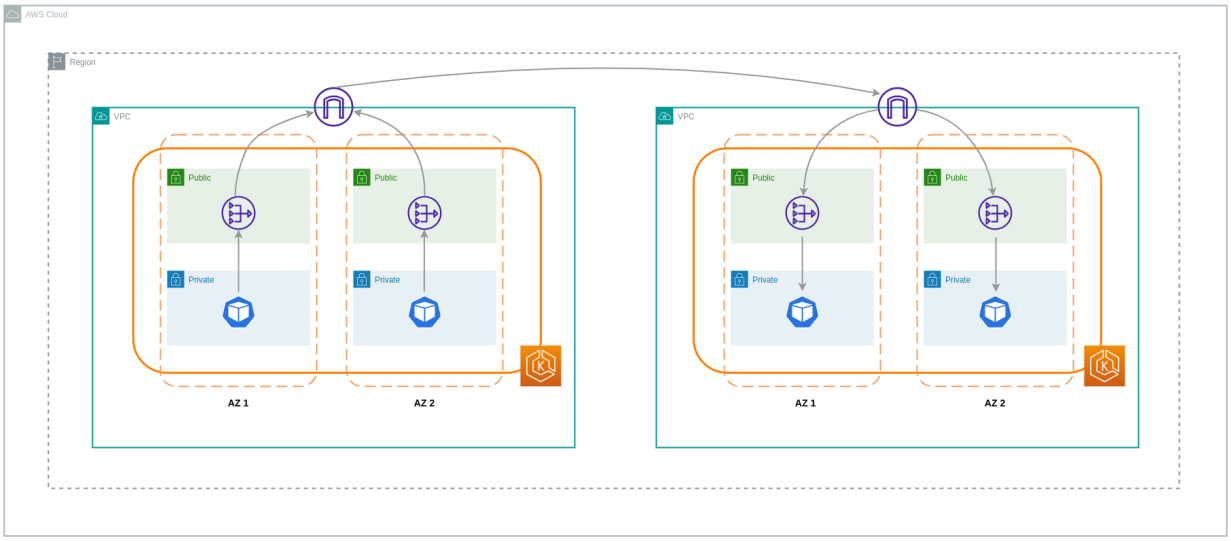
VPC 端點會按小時計費，並透過基礎 ENI 支付與資料處理相關的額外費用。請注意，AZ 間資料傳輸不會收費。

下圖顯示透過 VPC 端點與 AWS 服務通訊的 Pod。



VPCs之間的資料傳輸

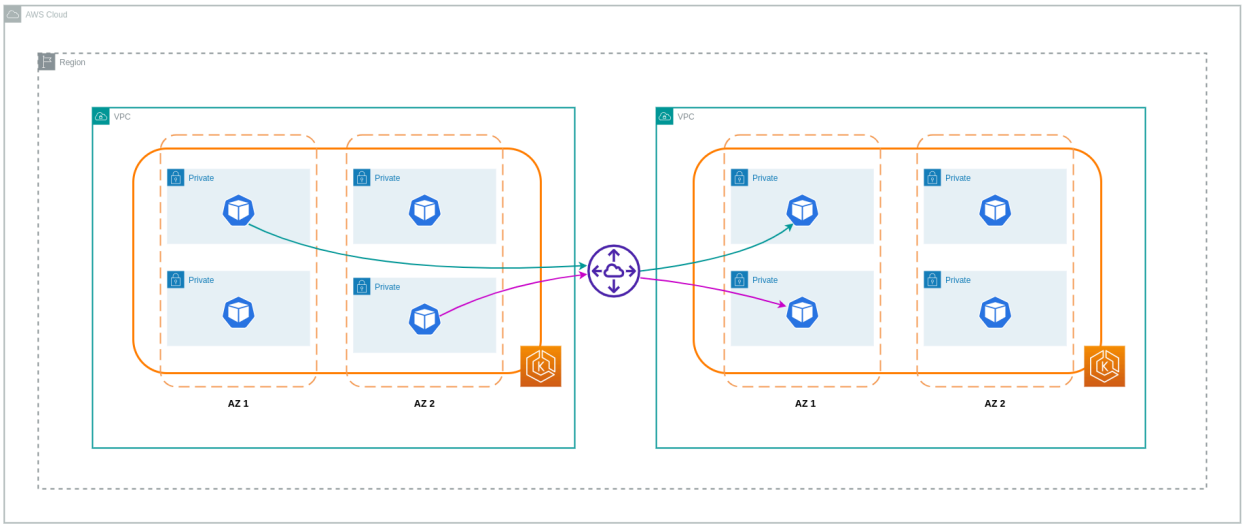
在某些情況下，您可能會在不同的 VPCs中（在同一 AWS 區域內）有工作負載，需要彼此通訊。這可以透過連接到個別 VPCs 的網際網路閘道，允許流量周遊公有網際網路來完成。這類通訊可以透過在公有子網路中部署基礎設施元件來啟用，例如 EC2 執行個體、NAT Gateway 或 NAT 執行個體。不過，包含這些元件的設定將產生處理/傳輸資料進出 VPCs的費用。如果往返個別 VPCs流量在 AZs 之間移動，則資料傳輸會產生額外費用。下圖說明使用 NAT Gateway 和網際網路閘道在不同 VPCs 中的工作負載之間建立通訊的設定。



VPC 互連連線

若要降低此類使用案例的成本，您可以使用 VPC 對等互連。使用 VPC 對等互連時，保留在相同可用區域內的網路流量無需支付資料傳輸費用。如果流量超過 AZs，會產生費用。不過，建議採用 VPC 對等互連方法來在相同 AWS 區域內不同 VPCs 中的工作負載之間進行符合成本效益的通訊。不過，請務必注意，VPC 對等互連主要對 1：1 VPC 連線有效，因為它不允許傳輸式聯網。

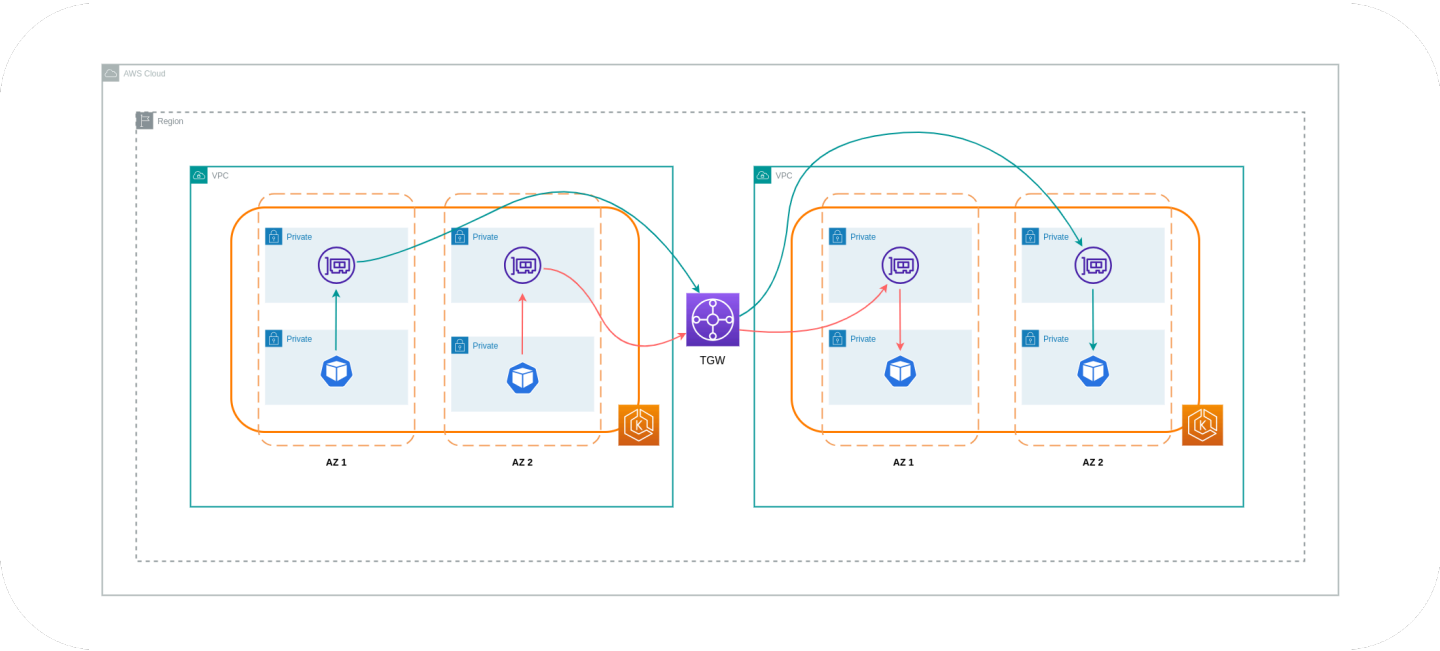
下圖是透過 VPC 互連連線進行工作負載通訊的高階表示。



暫時性網路連線

如上一節所述，VPC 對等連線不允許傳輸網路連線。如果您想要將 3 個以上的 VPCs 與傳輸性聯網需求連接，則應使用 Transit Gateway (TGW)。這可讓您克服 VPC 對等互連的限制，或與在多個 VPC 之間具有多個 VPC 對等互連相關聯的任何操作額外負荷 VPCs 。對於傳送至 TGW 的資料，您需要按小時計費。沒有與流經 TGW 的可用區域間流量相關聯的目的地成本。

下圖顯示在不同 VPCs但在相同 AWS 區域內流經 TGW 的可用區域間流量。



使用服務網格

服務網格提供強大的聯網功能，可用於降低 EKS 叢集環境中的網路相關成本。不過，如果您採用，您應該仔細考慮服務網格將對您的環境帶來的操作任務和複雜性。

將流量限制為可用區域

使用 Istio 的區域權重分佈

Istio 可讓您在路由發生後將網路政策套用至流量。這是使用目的地規則來完成的，例如地區性加權分佈。使用此功能，您可以控制流量的權重（以百分比表示），而流量可根據其原始伺服器前往特定目的地。此流量的來源可以是來自外部（或面向公有）負載平衡器或叢集本身內的 Pod。當所有 Pod 端點都可用時，將根據加權循環配置負載平衡演算法選取地區性。如果某些端點運作狀態不佳或無法使用，則會自動調整地區權重，以反映可用端點中的此變更。

> **Note**
>
> 在實作地區性加權分佈之前，您應該先了解網路流量模式，以及目的地規則政策可能對應用程式行為造成的影響。因此，請務必使用 AWS X-Ray 或 Jaeger 等工具建立分散式追蹤機制。

上述 Istio 目的地規則也可以套用，以管理從負載平衡器到 EKS 叢集中 Pod 的流量。地區加權分佈規則可以套用至從高可用性負載平衡器（特別是輸入閘道）接收流量的服務。這些規則可讓您根據區域原始伺服器 - 在這種情況下的負載平衡器，控制流向何處的流量。如果設定正確，相較於將流量平均或隨機分配到不同AZs Pod 複本的負載平衡器，產生的輸出跨區域流量較少。

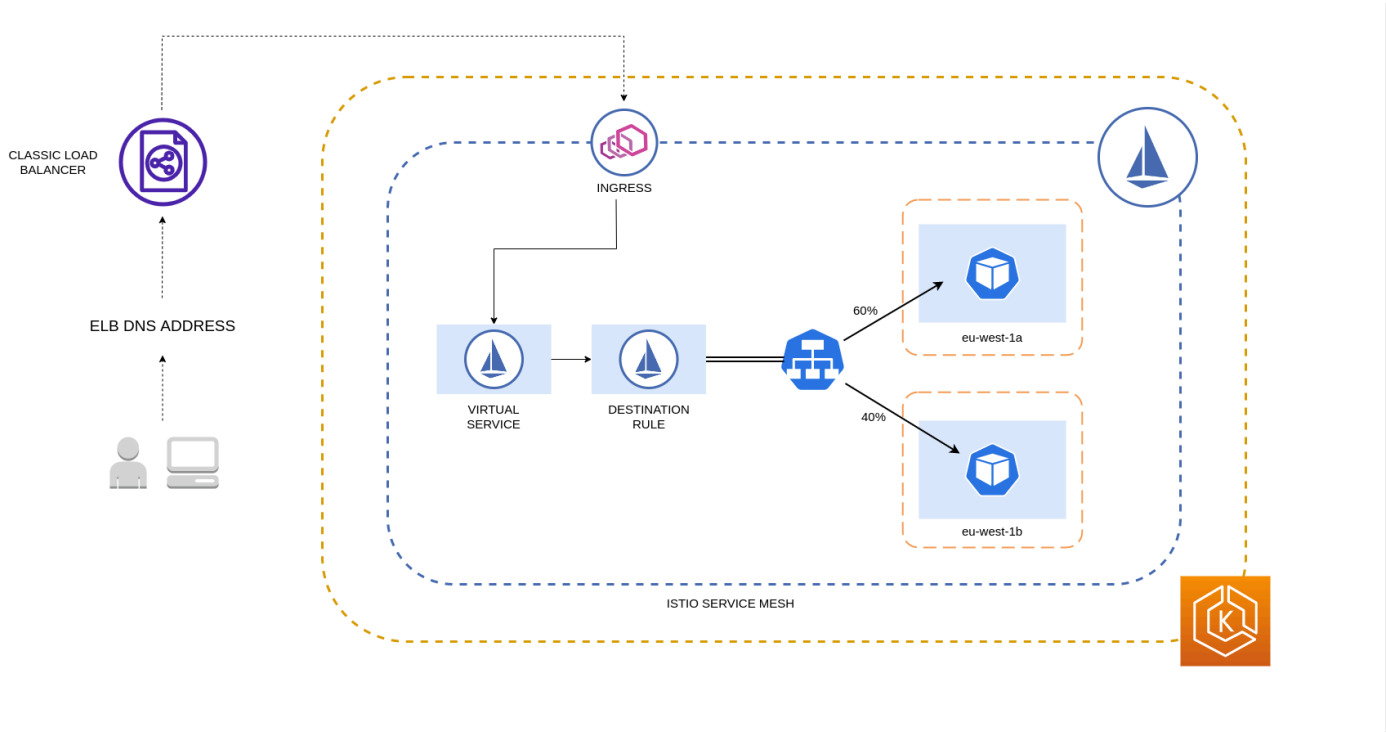
以下是 Istio 中目的地規則資源的程式碼區塊範例。如下方所示，此資源會指定區域中 3 個不同AZs傳入流量的加權組態eu-west-1。這些組態宣告，來自指定 AZ 的大部分傳入流量（在此案例中為 70%) 應代理至其來源相同 AZ 中的目的地。

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: express-test-dr
spec:
  host: express-test.default.svc.cluster.local
  trafficPolicy:
    loadBalancer:                      +
      localityLbSetting:
        distribute:
        - from: eu-west-1/eu-west-1a/  +
          to:
            "eu-west-1/eu-west-1a/_": 70
            "eu-west-1/eu-west-1b/_": 20
            "eu-west-1/eu-west-1c/_": 10
        - from: eu-west-1/eu-west-1b/_  +
          to:
            "eu-west-1/eu-west-1a/_": 20
            "eu-west-1/eu-west-1b/_": 70
            "eu-west-1/eu-west-1c/_": 10
        - from: eu-west-1/eu-west-1c/_  +
          to:
            "eu-west-1/eu-west-1a/_": 20
            "eu-west-1/eu-west-1b/_": 10
            "eu-west-1/eu-west-1c/*": 70**
    connectionPool:
      http:
```

```
      http2MaxRequests: 10
      maxRequestsPerConnection: 10
  outlierDetection:
    consecutiveGatewayErrors: 1
    interval: 1m
    baseEjectionTime: 30s
```

> **Note**
>
> 可分佈目的地的最小權重為 1%。這是因為在主要目的地中的端點運作狀態不佳或無法使用的情況下，維護容錯移轉區域和區域。

下圖說明 eu-west-1 區域中具有高度可用負載平衡器的情況，並套用地區權重分佈。此圖表的目的地規則政策設定為將 60% 來自 eu-west-1a 的流量傳送至相同 AZ 中的 Pod，而 40% 來自 eu-west-1a 的流量應傳送至 eu-west-1b 中的 Pod。



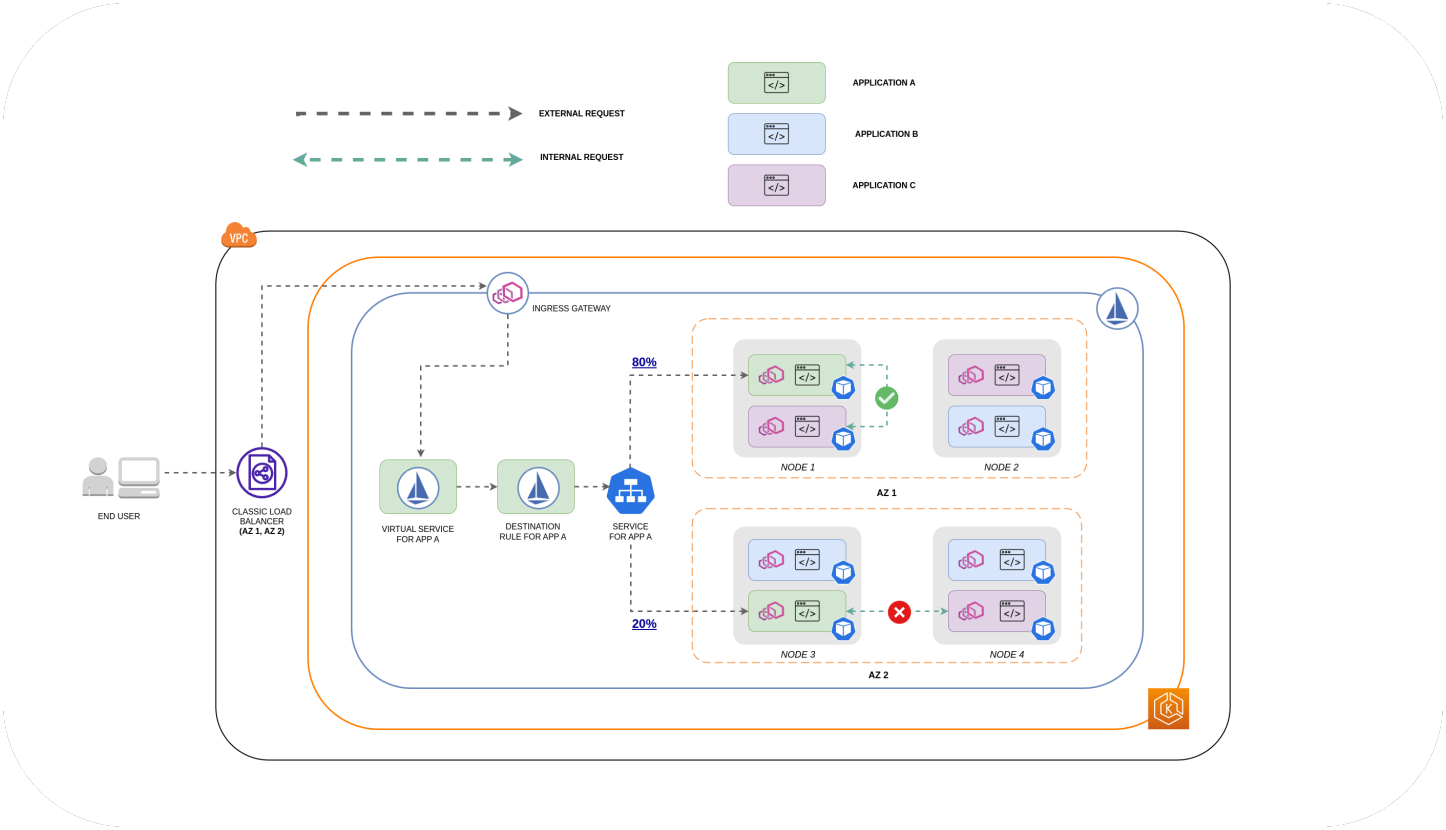
將流量限制為可用區域和節點

搭配 Istio 使用服務內部流量政策

若要降低與外部傳入流量和 Pod 之間的內部流量相關的網路成本，您可以結合 Istio 的目的地規則和 Kubernetes Service 內部流量政策。將 Istio 目的地規則與服務內部流量政策結合的方式，主要取決於 3 件事：

- 微服務的角色
- 跨微服務的網路流量模式
- 應如何跨 Kubernetes 叢集拓撲部署微服務

下圖顯示巢狀請求的網路流程會是什麼樣子，以及上述政策如何控制流量。



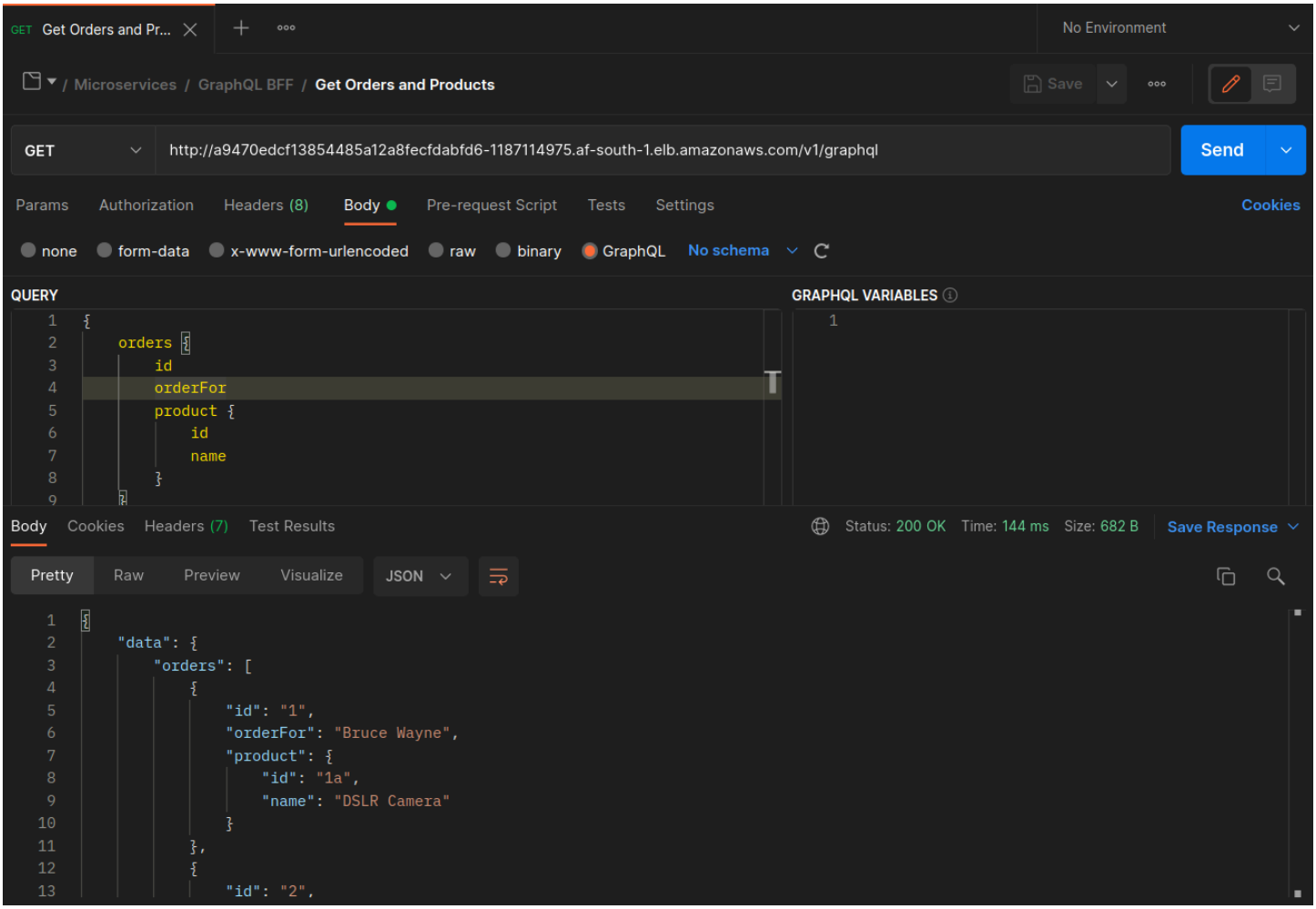
1. 最終使用者向 APP A 提出請求，進而向 APP C 提出巢狀請求。 此請求會先傳送至高可用性負載平衡器，其在 AZ 1 和 AZ 2 中具有執行個體，如上圖所示。
2. 外部傳入請求接著會由 Istio Virtual Service 路由至正確的目的地。
3. 請求路由後，Istio 目的地規則會根據來源 (AZs 1 或 AZ 2) 控制流向個別 AZ 的流量。
4. 接著流量會移至 Service for APP A，然後代理至個別的 Pod 端點。如圖所示，80% 的傳入流量會傳送至 AZ 1 中的 Pod 端點，而 20% 的傳入流量會傳送至 AZ 2。
5. 然後，APP A 向 APP C 提出內部請求。 APP C 的服務已啟用內部流量政策 (`internalTrafficPolicy`：Local`)。

6. 由於 APP C 可用的節點本機端點，從 APP A (NODE 1) 到 APP C 的內部請求成功。
7. 從 APP A (NODE 3) 到 APP C 的內部請求失敗，因為 APP C 沒有可用的節點本機端點。 如圖所示，APP C 在 NODE 3 上沒有複本。 **

以下螢幕擷取畫面是從此方法的即時範例擷取。第一組螢幕擷取畫面示範對 的成功外部請求，graphql以及從 成功巢狀請求graphql到節點 上共置orders複本ip-10-0-0-151.af-south-1.compute.internal。

test-results.txt

```
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-0-147.af-south-1.compute.i
NAME                        READY   STATUS    RESTARTS   AGE   IP          NODE
products-8567b458c8-brgdp   2/2     Running   0          64m   10.0.0.83   ip-10-0-0-147.af-south-1.com
--------------------------------------------------------------------------------
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-1-153.af-south-1.compute.i
NAME                        READY   STATUS    RESTARTS   AGE   IP         NODE
products-8567b458c8-5shq8   2/2     Running   0          65m   10.0.1.4   ip-10-0-1-153.af-south-1.comp
--------------------------------------------------------------------------------
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-1-68.af-south-1.compute.in
NAME                      READY   STATUS    RESTARTS   AGE   IP           NODE
orders-684bfbd6d9-5dxcf   2/2     Running   0          22m   10.0.1.145   ip-10-0-1-68.af-south-1.compu
--------------------------------------------------------------------------------
kubectl get pods -n ecommerce -o wide --field-selector spec.nodeName=ip-10-0-0-151.af-south-1.compute.i
NAME                      READY   STATUS    RESTARTS   AGE   IP           NODE
graphql-8f877c686-k7k9b   2/2     Running   0          18m   10.0.0.239   ip-10-0-0-151.af-south-1.comp
orders-684bfbd6d9-rc2zp   2/2     Running   0          18m   10.0.0.182   ip-10-0-0-151.af-south-1.comp
```



使用 Istio，您可以驗證和匯出代理知道的任何上游叢集和端點的統計資料。這有助於提供網路流程的影像，以及工作負載服務之間的分佈共用。繼續使用相同的範例，可以使用下列命令來取得graphql代理知道的orders端點：

```
kubectl exec -it deploy/graphql -n ecommerce -c istio-proxy -- curl localhost:15000/clusters | grep orders
```

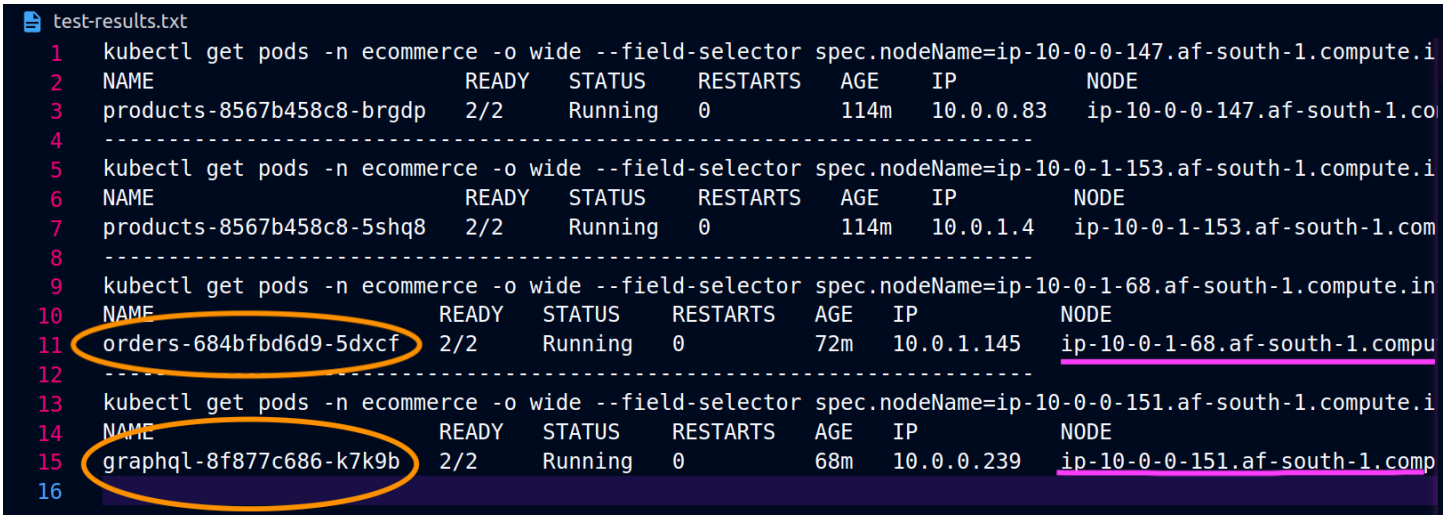
```
...
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_error::0**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_success::119**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_timeout::0**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**rq_total::119**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**health_flags::healthy**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**region::af-south-1**
orders-service.ecommerce.svc.cluster.local::10.0.1.33:3003::**zone::af-south-1b**
...
```

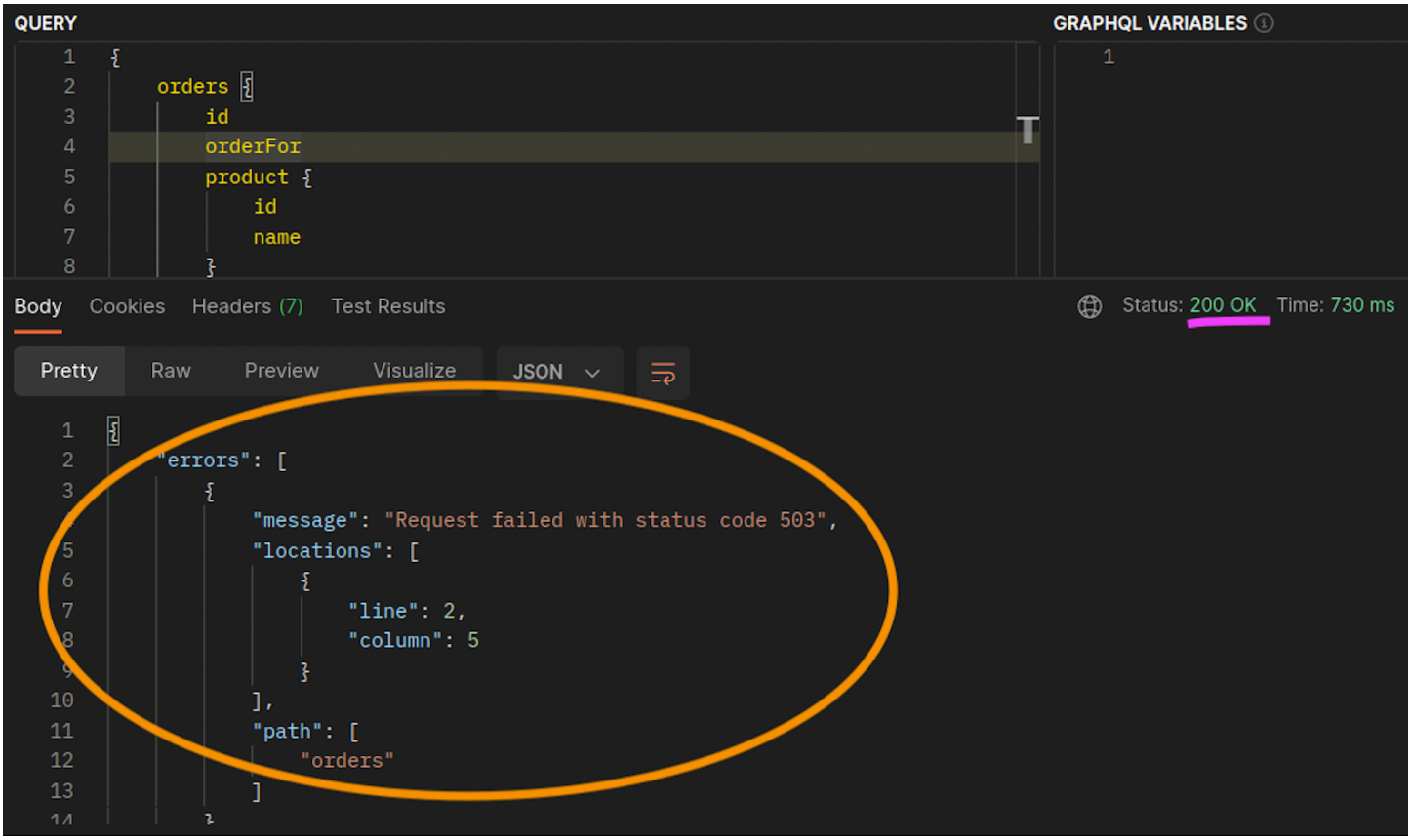
在此情況下，graphql代理只會知道其共用節點之複本的orders端點。如果您從訂單 Service 中移除internalTrafficPolicy: Local設定，並重新執行類似上述的命令，則結果會傳回分散在不同節點的複本的所有端點。此外，透過檢查個別端點rq_total的 ，您會注意到網路分佈中相對均勻的共用。因此，如果端點與在不同 AZs 中執行的上游服務相關聯，則跨區域的此網路分佈將產生更高的成本。

如上一節所述，您可以透過使用 Pod 親和性來共同定位經常通訊的 Pod。

```
...
spec:
...
  template:
    metadata:
      labels:
        app: graphql
        role: api
        workload: ecommerce
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - orders
            topologyKey: "kubernetes.io/hostname"
      nodeSelector:
        managedBy: karpenter
        billing-team: ecommerce
...
```

當 graphql和 orders 複本不存在於相同的節點 (ip-10-0-0-151.af-south-1.compute.internal) 200 response code時， 的第一個請求graphql會成功，如下方 Postman 螢幕擷取畫面中的 所述，而來自 graphql的第二個巢狀請求會因 而orders失敗503 response code。





其他資源

- 使用 Istio 處理 EKS 上的延遲和資料傳輸成本
- 探索拓撲感知提示對 Amazon Elastic Kubernetes Service 網路流量的影響
- 取得 Amazon EKS 跨可用區域 Pod 到 Pod 網路位元組的可見性
- 使用 Istio 最佳化 AZ 流量

- 使用拓撲感知路由最佳化 AZ 流量
- 使用服務內部流量政策最佳化 Kubernetes 成本和效能
- 使用 Istio 和服務內部流量政策最佳化 Kubernetes 成本和效能
- 常見架構的資料傳輸成本概觀
- 了解 AWS 容器服務的資料傳輸成本

儲存

概觀

在某些情況下，您可能想要執行需要短期或長期保留資料的應用程式。對於此類使用案例， Pod 可以定義和掛載磁碟區，以便其容器可以利用不同的儲存機制。Kubernetes 支援不同類型的磁碟區，用於暫時性和持久性儲存。儲存體的選擇主要取決於應用程式需求。對於每種方法，都會有成本影響，以下詳述的實務將協助您在 EKS 環境中需要某種形式的儲存體的工作負載實現成本效益。

暫時性容積

暫時性磁碟區適用於需要暫時性本機磁碟區，但不需要在重新啟動後保留資料的應用程式。此範例包括暫存空間、快取和唯讀輸入資料的需求，例如組態資料和秘密。您可以在此處找到 Kubernetes 暫時性磁碟區的詳細資訊。大多數暫時性磁碟區（例如 emptyDir、configMap、downwardAPI、秘密、hostpath) 都由本機連接的可寫入裝置（通常是根磁碟）或 RAM 提供支援，因此請務必選擇最具成本效益且效能最佳的主機磁碟區。

使用 EBS 磁碟區

我們建議從 gp3 開始做為主機根磁碟區。這是 Amazon EBS 提供的最新一般用途 SSD 磁碟區，相較於 gp2 磁碟區，它也提供較低的每 GB 價格（最多 20%)。

使用 Amazon EC2 執行個體存放區

Amazon EC2 執行個體存放區為您的 EC2 執行個體提供暫時區塊層級儲存。EC2 執行個體存放區提供的儲存體可透過實體連接至主機的磁碟存取。與 Amazon EBS 不同，您只能在執行個體啟動時連接執行個體存放區磁碟區，而且這些磁碟區只在執行個體的生命週期內存在。它們無法分離並重新連接到其他執行個體。您可以在這裡進一步了解 Amazon EC2 執行個體存放區。執行個體存放區磁碟區沒有相關的額外費用。這使得它們（執行個體存放區磁碟區）比具有大型 EBS 磁碟區的一般 EC2 執行個體更具成本效益。

若要在 Kubernetes 中使用本機儲存磁碟區，您應該使用 Amazon EC2 使用者資料分割、設定和格式化磁碟區，以便在 Pod 規格中將磁碟區掛載為 HostPath。或者，您可以利用本機持久性磁碟區靜態佈建器來簡化本機儲存管理。本機持久性磁碟區靜態佈建器可讓您透過標準 Kubernetes PersistentVolumeClaim (PVC) 界面存取本機執行個體存放區磁碟區。此外，它會佈建 PersistentVolumes (PVs)，其中包含節點親和性資訊，將 Pod 排程至正確的節點。雖然它使用 Kubernetes PersistentVolumes，但 EC2 執行個體存放區磁碟區本質上是暫時性的。寫入暫時性磁碟的資料只能在執行個體的生命週期內使用。當執行個體終止時，資料也是。如需詳細資訊，請參閱此部落格。

請記住，使用 Amazon EC2 執行個體存放區磁碟區時，總 IOPS 限制會與主機共用，並將 Pod 繫結至特定主機。在採用 Amazon EC2 執行個體存放區磁碟區之前，您應該徹底檢閱工作負載需求。

持久性磁碟區

Kubernetes 通常與執行中無狀態應用程式相關聯。不過，在某些情況下，您可能想要執行微服務，而這些微服務需要保留從一個請求到下一個請求的持久性資料或資訊。資料庫是這類使用案例的常見範例。不過，Pod 和其中的容器或程序本質上是暫時性的。若要將資料保留超過 Pod 的生命週期，您可以使用 PVs 定義在獨立於 Pod 的特定位置對儲存體的存取。與 PVs 相關的成本高度取決於使用的儲存體類型，以及應用程式如何使用它。

這裡列出支援 Amazon EKS 上 Kubernetes PVs 的不同儲存選項類型。以下涵蓋的儲存選項包括 Amazon EBS、Amazon EFS、Amazon FSx for Lustre、Amazon FSx for NetApp ONTAP。

Amazon Elastic Block Store (EBS) 磁碟區

Amazon EBS 磁碟區可作為 Kubernetes PVs 使用，以提供區塊層級儲存磁碟區。這些非常適合依賴隨機讀取和寫入的資料庫，以及執行長時間連續讀取和寫入的輸送量密集型應用程式。Amazon Elastic Block Store Container Storage Interface (CSI) 驅動程式可讓 Amazon EKS 叢集管理持久性磁碟區的 Amazon EBS 磁碟區的生命週期。容器儲存界面可啟用並促進 Kubernetes 與儲存系統之間的互動。當 CSI 驅動程式部署到您的 EKS 叢集時，您可以透過原生 Kubernetes 儲存資源存取其功能，例如持久性磁碟區 (PVs)、持久性磁碟區宣告 (PVCs) 和儲存類別 SCs)。此連結提供如何與 Amazon EBS CSI 驅動程式互動的實用範例。

選擇正確的磁碟區

我們建議您使用最新一代的區塊儲存 (gp3)，因為它可在價格和效能之間取得適當的平衡。它還可讓您獨立擴展磁碟區 IOPS 和輸送量，而不需要佈建額外的區塊儲存容量。如果您目前正在使用 gp2 磁碟區，強烈建議遷移至 gp3 磁碟區。部落格文章將 Amazon EKS 叢集從 gp2 遷移至 gp3 EBS 磁碟區，說明如何使用需要應用程式停機時間的 CSI 磁碟區快照功能，在 Amazon EKS 叢集上使用備份和還原從 gp3 上的 gp2 遷移。

Amazon EBS 允許變更線上磁碟區大小、IOPS 和輸送量等磁碟區特性。利用此功能，即可使用此部落格中所述的任一 PVC 註釋從 gp3 上的 gp2 遷移，而無需應用程式停機，這需要 EBS CSI 驅動程式 v1.19.0+，或從 Amazon EKS v1.31 和 EBS CSI 驅動程式 1.35 開始，方法是使用此處所述的 VolumeAttributesClass API。

當您的應用程式需要更高的效能，且所需的磁碟區大於單一 gp3 磁碟區可支援的磁碟區時，您應該考慮使用 io2 區塊表達式。這種類型的儲存體非常適合您最大、最密集的 I/O 和關鍵任務部署，例如 SAP HANA 或其他低延遲需求的大型資料庫。請記住，執行個體的 EBS 效能受到執行個體效能限制的限制，因此並非所有執行個體都支援 io2 區塊表達磁碟區。您可以在此文件中檢查支援的執行個體類型和其他考量事項。

單一 gp3 磁碟區最多可支援 16，000 個最大 IOPS、1，000 MiB/s 最大輸送量、16TiB。最新一代的佈建 IOPS SSD 磁碟區，可提供高達 256，000 IOPS、4，000 MiB/s、輸送量和 64TiB。

在這些選項中，您應該根據應用程式的需求，量身打造最佳的儲存效能和成本。

隨著時間的推移進行監控和最佳化

請務必了解應用程式的基準效能，並針對選取的磁碟區進行監控，以檢查是否符合您的需求/期望，或是過度佈建（例如，佈建 IOPS 未充分利用的情況）。

您可以在累積資料時逐漸增加磁碟區的大小，而不是從頭開始配置大型磁碟區。您可以使用 Amazon Elastic Block Store CSI 驅動程式 (aws-ebs-csi-driver) 中的磁碟區調整大小功能動態調整磁碟區大小。aws-ebs-csi-driver 請記住，您只能增加 EBS 磁碟區大小。

若要識別和移除任何懸置的 EBS 磁碟區，您可以使用 AWS 信任顧問的成本最佳化類別。此功能可協助您識別一段時間內寫入活動極低的未連接磁碟區或磁碟區。有一個雲端原生的開放原始碼唯讀工具，稱為 Popeye，可掃描即時 Kubernetes 叢集，並報告已部署資源和組態的潛在問題。例如，它可以掃描未使用的 PVs和 PVCs並檢查它們是否繫結，或是否有任何磁碟區掛載錯誤。

如需監控的詳細資訊，請參閱 EKS 成本最佳化可觀測性指南。

您可以考慮的另一個選項是 AWS Compute Optimizer Amazon EBS 磁碟區建議。此工具會自動識別最佳磁碟區組態，以及所需的正確效能等級。例如，它可以根據過去 14 天內的最大使用率，用於與佈建 IOPS、磁碟區大小和 EBS 磁碟區類型相關的最佳設定。它也會量化從其建議衍生的潛在每月成本節省。您可以檢閱此部落格以取得更多詳細資訊。

備份保留政策

您可以透過拍攝point-in-time快照來備份 Amazon EBS 磁碟區上的資料。Amazon EBS CSI 驅動程式支援磁碟區快照。您可以使用此處概述的步驟，了解如何建立快照和還原 EBS PV。

後續快照是增量備份，這表示只會儲存最近快照之後在裝置上變更的區塊。如此無須複製所有資料，可大幅減少建立快照所需的時間，並節省儲存成本。不過，在沒有適當保留政策的情況下增加舊 EBS 快照的數量，可能會導致大規模操作時產生非預期的成本。如果您透過 AWS API 直接備份 Amazon EBS 磁碟區，則可以利用 Amazon Data Lifecycle Manager (DLM)，為 Amazon Elastic Block Store (EBS) 快照和 EBS 支援的 Amazon Machine Image (AMIs) 提供自動化的政策型生命週期管理解決方案。主控台可讓您更輕鬆地自動建立、保留和刪除 EBS 快照和 AMIs。

> **Note**
>
> 目前無法透過 Amazon EBS CSI 驅動程式使用 Amazon DLM。

在 Kubernetes 環境中，您可以利用名為 Velero 的開放原始碼工具來備份 EBS 持久性磁碟區。您可以在排程任務使備份過期時設定 TTL 旗標。以下是來自 Velero 的指南範例。

Amazon Elastic File System

Amazon Elastic File System (EFS) 是一種無伺服器、全彈性的檔案系統，可讓您使用標準檔案系統界面和檔案系統語意來共用檔案資料，以處理廣泛的工作負載和應用程式。工作負載和應用程式的範例包括 Wordpress 和 Drupal、JIRA 和 Git 等開發人員工具，以及 Jupyter 等共用筆記本系統以及主目錄。

Amazon EFS 的主要優點之一是，它可以由分散在多個節點和多個可用區域的多個容器掛載。另一個好處是您只需為所使用的儲存體付費。當您新增和移除不需要容量規劃的檔案時，EFS 檔案系統會自動成長和縮減。

若要在 Kubernetes 中使用 Amazon EFS，您需要使用 Amazon Elastic File System Container Storage Interface (CSI) Driver， aws-efs-csi-driver。目前，驅動程式可以動態建立存取點。不過，必須先佈建 Amazon EFS 檔案系統，並提供 做為 Kubernetes 儲存類別參數的輸入。

選擇正確的 EFS 儲存類別

Amazon EFS 提供四個儲存類別。

兩個標準儲存類別：

- Amazon EFS 標準
- Amazon EFS Standard-Infrequent Access (EFS Standard-IA)

兩個單區域儲存類別：

- Amazon EFS 單區域
- Amazon EFS One Zone-Infrequent Access (EFS One Zone-IA)

不常存取 (IA) 儲存類別針對未每天存取的檔案進行成本最佳化。透過 Amazon EFS 生命週期管理，您可以將生命週期政策期間 (7、14、30、60 或 90 天） 未存取的檔案移至 IA 儲存類別，相較於 EFS Standard 和 EFS One Zone 儲存類別，這些儲存類別分別可將儲存成本降低高達 92%。

透過 EFS Intelligent-Tiering，生命週期管理會監控檔案系統的存取模式，並自動將檔案移至最佳的儲存類別。

> **Note**
>
> aws-efs-csi-driver 目前無法控制變更儲存類別、生命週期管理或 Intelligent-Tiering。這些應該在 AWS 主控台或透過 EFS APIs 手動設定。

> **Note**
>
> aws-efs-csi-driver 與以視窗為基礎的容器映像不相容。

> **Note**
>
> 由於 DiskUsage 函數會耗用與檔案系統大小成比例的記憶體量，因此啟用 vol-metrics-opt-in （發射磁碟區指標） 時，存在已知的記憶體問題。目前，我們建議在大型檔案系統上停用 `--vol-metrics-opt-in` 選項，以避免耗用太多記憶體。以下是 github 問題連結，以取得更多詳細資訊。

Amazon FSx for Lustre

Lustre 是一種高效能平行檔案系統，通常用於需要高達數百 GB/秒輸送量和每次操作延遲低於毫秒的工作負載。它用於機器學習訓練、財務建模、HPC 和影片處理等案例。Amazon FSx for Lustre 提供具有可擴展性和效能的全受管共用儲存體，與 Amazon S3 無縫整合。

您可以使用 FSx for Lustre 支援的 Kubernetes 持久性儲存磁碟區，使用 Amazon EKS 的 FSx for Lustre CSI 驅動程式或 AWS 上的自我管理 Kubernetes 叢集。如需詳細資訊和範例，請參閱 Amazon EKS 文件。

Amazon S3 的連結

建議將位於 Amazon S3 上的高耐用性長期資料儲存庫與您的 FSx for Lustre 檔案系統連結。一旦連結，大型資料集會視需要從 Amazon S3 延遲載入至 FSx for Lustre 檔案系統。您也可以執行分析並將結果傳回 S3，然後刪除 Lustre 檔案系統。

選擇正確的部署和儲存選項

FSx for Lustre 提供不同的部署選項。第一個選項稱為暫存，不會複寫資料，而第二個選項稱為持久性，顧名思義，會保留資料。

第一個選項（速記) 可用來降低暫時性短期資料處理的成本。持久性部署選項是專為長期儲存而設計，可自動複寫 AWS 可用區域內的資料。它還支援 SSD 和 HDD 儲存。

您可以在 FSx 中的參數下，為 lustre 檔案系統的 Kubernetes StorageClass 設定所需的部署類型。以下是提供範例範本的連結。

> **Note**
>
> 對於對延遲敏感的工作負載或需要最高 IOPS/輸送量層級的工作負載，您應該選擇 SSD 儲存。對於不對延遲敏感的以輸送量為中心的工作負載，您應該選擇 HDD 儲存。

啟用資料壓縮

您也可以指定「LZ4」為資料壓縮類型，在檔案系統上啟用資料壓縮。啟用後，所有新寫入的檔案都會在 FSx for Lustre 上自動壓縮，再寫入磁碟，並在讀取時解壓縮。LZ4 資料壓縮演算法無失真，因此原始資料可以從壓縮資料完全重建。

您可以在 FSx 中 lustre 檔案系統的 Kubernetes StorageClass 的參數下，將資料壓縮類型設定為 LZ4。當值設定為 NONE 時，壓縮會停用，這是預設值。此連結提供範例範本。

> **Note**
>
> Amazon FSx for Lustre 與以視窗為基礎的容器映像不相容。

Amazon FSx for NetApp ONTAP

Amazon FSx for NetApp ONTAP 是建置在 NetApp ONTAP 檔案系統的全受管共用儲存體。FSx for ONTAP 提供功能豐富、快速且靈活的共用檔案儲存，可從 AWS 或內部部署中執行的 Linux、Windows 和 macOS 運算執行個體廣泛存取。

Amazon FSx for NetApp ONTAP 支援兩個儲存層：1/主要層和 2/容量集區層。

主要層是佈建的高效能 SSD 型層，用於主動、延遲敏感的資料。全彈性容量集區層針對不常存取的資料進行成本最佳化、隨著資料分層而自動擴展，並提供幾乎無限制的 PB 容量。您可以在容量集區儲存上啟用資料壓縮和重複資料刪除，並進一步減少資料使用的儲存容量。NetApp 的原生政策型 FabricPool 功能會持續監控資料存取模式，並在儲存層之間自動雙向傳輸資料，以最佳化效能和成本。

NetApp 的 Astra Trident 使用 CSI 驅動程式提供動態儲存協同運作，可讓 Amazon EKS 叢集管理 Amazon FSx for NetApp ONTAP 檔案系統支援的持久性磁碟區 PVs 生命週期。若要開始使用，請參閱 Astra Trident 文件中的使用 Astra Trident 與 Amazon FSx for NetApp ONTAP。

其他考量

最小化容器映像的大小

部署容器後，容器映像會以多層的形式快取在主機上。透過減少映像的大小，可以減少主機上所需的儲存量。

透過從頭開始使用縮減的基礎映像，例如暫存映像或無痕容器映像（僅包含您的應用程式及其執行時間相依性），除了減少攻擊表面積和縮短映像提取時間等輔助優點之外，您還可以降低儲存成本。

您也應該考慮使用開放原始碼工具，例如 Slim.ai，提供簡單、安全的方式來建立最少的映像。

多層套件、工具、應用程式相依性、程式庫可以輕鬆膨脹容器映像大小。透過使用多階段組建，您可以選擇性地將成品從一個階段複製到另一個階段，排除最終映像中不需要的所有項目。您可以在此處查看更多映像建置最佳實務。

要考慮的另一件事是保留快取映像的時間。使用特定磁碟量時，您可能想要清除映像快取中的過時映像。這樣做有助於確保您有足夠的空間進行主機操作。根據預設， kubelet 每五分鐘對未使用的映像和每分鐘未使用的容器執行垃圾收集。

若要為未使用的容器和映像垃圾收集設定選項，請使用組態檔案調整 kubelet，並使用 *KubeletConfiguration* 資源類型變更與垃圾收集相關的參數。

您可以在 Kubernetes 文件中進一步了解。

可觀測性

簡介

可觀測性工具可協助您有效率地偵測、修復和調查工作負載。遙測資料的成本會隨著您使用 EKS 而自然增加。有時，平衡您的營運需求並衡量對業務重要的事項，以及控制可觀測性成本可能很困難。本指南著重於可觀測性的三個支柱的成本最佳化策略：日誌、指標和追蹤。這些最佳實務都可以獨立套用，以符合組織的最佳化目標。

日誌

記錄在監控和故障診斷叢集中的應用程式時扮演重要角色。有多種策略可用來最佳化記錄成本。下列最佳實務策略包括檢查您的日誌保留政策，以實作日誌資料保留的時間長度、根據重要性將日誌資料傳送至不同的儲存選項，以及使用日誌篩選來縮小儲存日誌訊息的類型。有效管理日誌遙測可為您的環境節省成本。

EKS 控制平面

最佳化您的控制平面日誌

Kubernetes 控制平面是一組管理叢集的元件，這些元件會將不同類型的資訊做為日誌串流傳送至 Amazon CloudWatch 中的日誌群組。雖然啟用所有控制平面日誌類型都有好處，但您應該了解每個日誌中的資訊，以及存放所有日誌遙測的相關費用。您需支付從叢集傳送至 Amazon CloudWatch Logs 之日誌的標準 CloudWatch Logs 資料擷取和儲存成本。 Amazon CloudWatch 在啟用它們之前，請評估每個日誌串流是否必要。

例如，在非生產叢集中，選擇性地啟用特定日誌類型，例如 api 伺服器日誌，僅用於分析和之後停用。但是，對於生產叢集，您可能無法重現事件，並解決問題需要更多日誌資訊，則您可以啟用所有日誌類型。進一步的控制平面成本最佳化實作詳細資訊請參閱此部落格文章。

將日誌串流至 S3

另一個成本最佳化最佳實務是透過 CloudWatch Logs 訂閱將控制平面日誌串流至 S3。利用 CloudWatch Logs 訂閱可讓您選擇性地將日誌轉送至 S3，相較於在 CloudWatch 中無限期保留日誌，它可提供更具成本效益的長期儲存。例如，對於生產叢集，您可以建立關鍵日誌群組，並在 15 天後利用訂閱將這些日誌串流到 S3。這將確保您能夠快速存取日誌以供分析，但也可以將日誌移至更具成本效益的儲存體，以節省成本。

> **Important**
>
> 自 9/5/2023 起，EKS 日誌在 Amazon CloudWatch Logs 中歸類為 Vended Logs。已終止的日誌是特定 AWS 服務日誌，由 AWS 服務代表客戶原生發佈，並可用於磁碟區折扣定價。請造訪 Amazon CloudWatch 定價頁面，以進一步了解 Vended Logs 定價。

EKS 資料平面

日誌保留

Amazon CloudWatch 的預設保留政策是無限期保留日誌，永遠不會過期，因此產生適用於您 AWS 區域的儲存成本。為了降低儲存成本，您可以根據您的工作負載需求，為每個日誌群組自訂保留政策。

在開發環境中，可能不需要較長的保留期。但在生產環境中，您可以設定更長的保留政策，以滿足故障診斷、合規和容量規劃需求。例如，如果您在尖峰假日期間執行電子商務應用程式，系統負載會比較重，而且可能會出現可能無法立即注意到的問題，則建議您設定較長的日誌保留時間，以進行詳細的故障診斷和事件後分析。

您可以在 AWS CloudWatch 主控台或 AWS API 中設定保留期間，根據每個日誌群組，保留期間從 1 天到 10 年。擁有彈性的保留期間可以節省日誌儲存成本，同時維護關鍵日誌。

日誌儲存選項

儲存是可觀測性成本的一大驅動因素，因此最佳化日誌儲存策略至關重要。您的策略應與您的工作負載需求保持一致，同時維持效能和可擴展性。降低日誌儲存成本的一個策略是利用 AWS S3 儲存貯體及其不同的儲存層。

將日誌直接轉送至 S3

考慮將較不重要的日誌直接轉送至 S3，而非 Cloudwatch。這可能會對日誌儲存成本產生立即影響。其中一個選項是使用 Fluentbit 直接將日誌轉送至 S3。您可以在 [OUTPUT] 區段中定義此項目，也就是 FluentBit 傳輸容器日誌以進行保留的目的地。在此處檢閱其他組態參數。

```
[OUTPUT]
        Name eks_to_s3
        Match application.*
        bucket $S3_BUCKET name
        region us-east-2
        store_dir /var/log/fluentbit
```

```
        total_file_size 30M
        upload_timeout 3m
```

僅將日誌轉送至 CloudWatch 進行短期分析

如需更關鍵的日誌，例如您可能需要立即對資料執行分析的生產環境，請考慮將日誌轉送至 CloudWatch。您可以在 [OUTPUT] 區段中定義此項目，也就是 FluentBit 傳輸容器日誌以進行保留的目的地。在此處檢閱其他組態參數。

```
[OUTPUT]
        Name eks_to_cloudwatch_logs
        Match application.*
        region us-east-2
        log_group_name fluent-bit-cloudwatch
        log_stream_prefix from-fluent-bit-
        auto_create_group On
```

不過，這不會對您的成本節省造成立即影響。若要節省更多成本，您必須將這些日誌匯出至 Amazon S3。

從 CloudWatch 匯出至 Amazon S3

若要長期存放 Amazon CloudWatch logs，建議您將 Amazon EKS CloudWatch 日誌匯出至 Amazon Simple Storage Service (Amazon S3)。您可以透過主控台或 API 建立匯出任務，將日誌轉送至 Amazon S3 儲存貯體。完成後，Amazon S3 會呈現許多選項，以進一步降低成本。您可以定義自己的 Amazon S3 生命週期規則，將日誌移至符合您需求的儲存類別，或利用 Amazon S3 Intelligent-Tiering 儲存類別，讓 AWS 根據您的使用模式自動將資料移至長期儲存。如需詳細資訊，請參閱此部落格。例如，對於您的生產環境日誌， 會在 CloudWatch 中存放超過 30 天，然後匯出至 Amazon S3 儲存貯體。然後，如果您稍後需要參考日誌，您可以使用 Amazon Athena 查詢 Amazon S3 儲存貯體中的資料。

降低日誌層級

為您的應用程式練習選擇性記錄。根據預設，您的應用程式和節點都會輸出日誌。針對您的應用程式日誌，調整日誌層級以符合工作負載和環境的關鍵性。例如，以下 java 應用程式正在輸出INFO日誌，這是典型的預設應用程式組態，根據程式碼，可能會導致大量的日誌資料。

```
import org.apache.log4j.*;

public class LogClass {
```

```
   private static org.apache.log4j.Logger log = Logger.getLogger(LogClass.class);

public static void main(String[] args) {
      log.setLevel(Level.INFO);

   log.debug("This is a DEBUG message, check this out!");
   log.info("This is an INFO message, nothing to see here!");
   log.warn("This is a WARN message, investigate this!");
   log.error("This is an ERROR message, check this out!");
   log.fatal("This is a FATAL message, investigate this!");    } }
```

在開發環境中，將日誌層級變更為 DEBUG，因為這可協助您在問題進入生產環境之前進行偵錯或擷取潛在問題。

```
      log.setLevel(Level.DEBUG);
```

在生產環境中，請考慮將日誌層級修改為 ERROR或 FATAL。這只會在您的應用程式發生錯誤時輸出日誌、減少日誌輸出，並協助您專注於應用程式狀態的重要資料。

```
      log.setLevel(Level.ERROR);
```

您可以微調各種 Kubernetes 元件日誌層級。例如，如果您使用 Bottlerocket 做為 EKS 節點作業系統，則有組態設定可讓您調整 kubelet 程序日誌層級。此組態設定的程式碼片段如下。請注意預設日誌層級 2，其會調整kubelet程序的記錄詳細程度。

```
[settings.kubernetes]
log-level = "2"
image-gc-high-threshold-percent = "85"
image-gc-low-threshold-percent = "80"
```

對於開發環境，您可以設定大於 2 的日誌層級，以檢視其他事件，這非常適合偵錯。對於生產環境，您可以將閾卡設定為 0，以便僅檢視關鍵事件。

利用篩選條件

使用預設的 EKS Fluentbit 組態將容器日誌傳送至 Cloudwatch 時，FluentBit 會擷取並傳送充實 Kubernetes 中繼資料的所有應用程式容器日誌至 Cloudwatch，如以下[INPUT]組態區塊所示。

```
 [INPUT]
     Name                tail
```

```
    Tag                 application.*
    Exclude_Path        /var/log/containers/cloudwatch-agent*, /var/log/containers/
fluent-bit*, /var/log/containers/aws-node*, /var/log/containers/kube-proxy*
    Path                /var/log/containers/*.log
    Docker_Mode         On
    Docker_Mode_Flush   5
    Docker_Mode_Parser  container_firstline
    Parser              docker
    DB                  /var/fluent-bit/state/flb_container.db
    Mem_Buf_Limit       50MB
    Skip_Long_Lines     On
    Refresh_Interval    10
    Rotate_Wait         30
    storage.type        filesystem
    Read_from_Head      ${READ_FROM_HEAD}
```

上述[INPUT]區段正在擷取所有容器日誌。這可能會產生可能不需要的大量資料。篩選此資料可減少傳送至 CloudWatch 的日誌資料量，因此可降低成本。您可以將篩選條件套用到日誌，然後再輸出到 CloudWatch。Fluentbit 會在 [FILTER]區段中定義此項目。例如，篩選掉 Kubernetes 中繼資料，使其無法附加到日誌事件，可能會降低您的日誌磁碟區。

```
    [FILTER]
        Name                nest
        Match               application.*
        Operation           lift
        Nested_under        kubernetes
        Add_prefix          Kube.

    [FILTER]
        Name                modify
        Match               application.*
        Remove              Kube.<Metadata_1>
        Remove              Kube.<Metadata_2>
        Remove              Kube.<Metadata_3>

    [FILTER]
        Name                nest
        Match               application.*
        Operation           nest
        Wildcard            Kube.*
        Nested_under        kubernetes
        Remove_prefix       Kube.
```

指標

指標提供有關系統效能的寶貴資訊。透過將所有與系統相關或可用的資源指標合併到集中位置，您可以獲得比較和分析效能資料的功能。這種集中式方法可讓您做出更明智的策略決策，例如擴展或縮減資源。此外，指標在評估資源的運作狀態方面扮演重要角色，可讓您在必要時採取主動措施。一般而言，可觀測性成本會隨著遙測資料收集和保留而擴展。您可以實作以下幾個策略來降低指標遙測的成本：僅收集重要的指標、降低遙測資料的基數，以及微調遙測資料收集的精細程度。

監控重要事項，並僅收集您需要的項目

第一個降低成本策略是減少您正在收集的指標數量，進而降低保留成本。

1. 首先，從您和/或利益相關者的要求向後工作，以確定最重要的指標。每個人的成功指標都不同！知道什麼是好東西，並對其進行測量。
2. 考慮深入探索您支援的工作負載，並識別其關鍵效能指標 (KPIs) a.k.a 'Golden Signals'。這些應符合業務和利益相關者要求。使用 Amazon CloudWatch 和指標數學計算 SLIs、SLOs 和 SLAs，對於管理服務可靠性至關重要。遵循本指南中概述的最佳實務，以有效監控和維護 EKS 環境的效能。
3. 然後，繼續瀏覽不同層級的基礎設施，以連接 EKS 叢集、節點和其他基礎設施指標，並將其與工作負載 KPI 建立關聯。 KPIs 將您的業務指標和操作指標存放在系統中，您可以將它們關聯在一起，並根據觀察到的對兩者的影響得出結論。
4. EKS 從控制平面、叢集 kube-state-metrics、Pod 和節點公開指標。所有這些指標的相關性取決於您的需求，但您可能不需要跨不同層的每個指標。您可以使用此 EKS 基本指標指南做為基準，以監控 EKS 叢集和工作負載的整體運作狀態。

以下是 Prometheus 抓取組態的範例，其中我們使用 `relabel_config`來僅保留 kubelet 指標`metric_relabel_config`，並捨棄所有容器指標。

```
kubernetes_sd_configs:
- role: endpoints
  namespaces:
    names:
    - kube-system
bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
tls_config:
  insecure_skip_verify: true
relabel_configs:
- source_labels: [__meta_kubernetes_service_label_k8s_app]
  regex: kubelet
  action: keep
```

```
  metric_relabel_configs:
  - source_labels: [__name__]
    regex: container_(network_tcp_usage_total|network_udp_usage_total|tasks_state|
cpu_load_average_10s)
    action: drop
```

如適用，減少基數

基數是指資料值的唯一性及其特定指標集的維度（例如 Prometheus 標籤）。高基數指標有許多維度，每個維度指標組合都有更高的唯一性。較高的基數會導致較大的指標遙測資料大小和儲存需求，進而增加成本。

在下面的高基數範例中，我們看到 指標、延遲、 具有維度、RequestID、CustomerID 和 Service，而且每個維度都有許多唯一的值。Cardinality 是每個維度可能值的組合。在 Prometheus 中，每組唯一維度/標籤都視為新的指標，因此高基數表示更多的指標。

在每個指標具有許多指標和維度/標籤的 EKS 環境中（叢集、命名空間、服務、Pod、容器等），基數通常會增加。為了最佳化成本，請仔細考慮您正在收集的指標基數。例如，如果您要彙總叢集層級視覺化的特定指標，則可以捨棄位於較低圖層的其他標籤，例如命名空間標籤。

為了識別 prometheus 中的高基數指標，您可以執行下列 PROMQL 查詢，以判斷哪些抓取目標具有最多指標（基數）：

```
topk_max(5, max_over_time(scrape_samples_scraped[1h]))
```

和下列 PROMQL 查詢可協助您判斷哪些抓取目標具有最高的指標流失（在指定的抓取中建立了多少個新指標系列）率：

```
topk_max(5, max_over_time(scrape_series_added[1h]))
```

如果您使用的是 grafana，則可以使用 Grafana Lab 的 Mimirtool 來分析您的 grafana 儀表板和 prometheus 規則，以識別未使用的高基數指標。請遵循本指南，了解如何使用 `mimirtool analyze`和 `mimirtool analyze prometheus`命令來識別儀表板中未參考的作用中指標。

考慮指標精細度

以更高的精細度收集指標，例如每秒與每分鐘，可能會對收集和存放多少遙測產生重大影響，進而增加成本。判斷合理的抓取或指標收集間隔，在足夠的精細程度之間取得平衡，以查看暫時性問題，並降低到足以符合成本效益的程度。減少用於容量規劃和較大時段分析的指標精細度。

以下是預設 AWS Distro for Opentelemetry (ADOT) EKS 附加元件收集器組態的程式碼片段。

> **Important**
>
> 全域 Prometheus 抓取間隔設定為 15 秒。此抓取間隔可能會增加，導致 Prometheus 中收集的指標資料量減少。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: my-collector-amp

...

config: |
    extensions:
      sigv4auth:
        region: "+++<YOUR_AWS_REGION>+++" service: "aps"+++</YOUR_AWS_REGION>+++

 receivers:
   #
   # Scrape configuration for the Prometheus Receiver
   # This is the same configuration used when Prometheus is installed using the
 community Helm chart
   #
   prometheus:
     config:
       global:   scrape_interval: 15s
         scrape_timeout: 10s
```

追蹤

與追蹤相關的主要成本來自追蹤儲存產生。透過追蹤，目標是收集足夠的資料來診斷和了解效能層面。不過，由於 X-Ray 追蹤成本是以轉送至 X-Ray 的資料為基礎，因此在轉送之後清除追蹤不會降低您的成本。讓我們檢閱降低追蹤成本的方法，同時維護資料以執行適當的分析。

套用取樣規則

根據預設，X-Ray 取樣率是保守的。定義抽樣規則，您可以在其中控制收集的資料量。這將提高效能效率，同時降低成本。透過降低取樣率，您可以從請求中收集追蹤，只需工作負載所需的項目，同時維持較低的成本結構。

例如，您有 java 應用程式，您想要偵錯 1 個有問題路由之所有請求的追蹤。

透過 SDK 設定 ，從 JSON 文件載入抽樣規則

```
{
"version": 2,
  "rules": [
    {
"description": "debug-eks",
      "host": "*",
      "http_method": "PUT",
      "url_path": "/history/*",
      "fixed_target": 0,
      "rate": 1,
      "service_type": "debug-eks"
    }
  ],
  "default": {
"fixed_target": 1,
    "rate": 0.1
  }
}
```

透過主控台

使用 AWS Distro for OpenTelemetry (ADOT) 套用尾部取樣

ADOT Tail Sampling 可讓您控制服務中擷取的追蹤量。不過， Tail Sampling 可讓您在請求中的所有範圍都完成後定義抽樣政策，而不是從頭開始。這進一步限制了傳輸到 CloudWatch 的原始資料量，進而降低成本。

例如，如果您要取樣 1% 的流量到登陸頁面，以及 10% 的請求到付款頁面，這可能會在 30 分鐘內讓您有 300 個追蹤。使用篩選特定錯誤的 ADOT Tail 取樣規則，您可以保留 200 個追蹤，以減少儲存的追蹤數量。

```
processors:
```

```
  groupbytrace:
    wait_duration: 10s
    num_traces: 300
    tail_sampling:
    decision_wait: 1s # This value should be smaller than wait_duration
    policies:
      - ..... # Applicable policies**
  batch/tracesampling:
    timeout: 0s # No need to wait more since this will happen in previous processors
    send_batch_max_size: 8196 # This will still allow us to limit the size of the
 batches sent to subsequent exporters

service:
  pipelines:
    traces/tailsampling:
      receivers: [otlp]
      processors: [groupbytrace, tail_sampling, batch/tracesampling]
      exporters: [awsxray]
```

利用 Amazon S3 Storage 選項

您應該利用 AWS S3 儲存貯體及其不同的儲存類別來存放追蹤。在保留期間到期之前，將追蹤匯出至 S3。使用 Amazon S3 生命週期規則，將追蹤資料移至符合您需求的儲存類別。

例如，如果您有 90 天的追蹤，Amazon S3 Intelligent-Tiering 可以根據您的使用模式自動將資料移至長期儲存。如果您需要稍後再參考追蹤，您可以使用 Amazon Athena 查詢 Amazon S3 中的資料。這可以進一步降低分散式追蹤的成本。

其他資源：

- 可觀測性最佳實務指南
- 最佳實務指標集合
- AWS re：Invent 2022 - Amazon (COP343) 的可觀測性最佳實務
- AWS re：Invent 2022 - 可觀測性：現代應用程式的最佳實務 (COP344)

Windows 的最佳實務

本指南提供有關執行 Windows 容器和節點的建議。

主題

- Amazon EKS 最佳化 Windows AMI 管理
- 為 Windows Pod 和容器設定 gMSA
- Windows 工作者節點強化
- 容器映像掃描
- Windows Server 版本和授權
- 記錄
- 監控
- Windows 網路
- 避免 OOM 錯誤
- 修補 Windows 伺服器和容器
- 執行異質工作負載
- Pod 安全內容
- 持久性儲存選項
- 強化 Windows 容器映像

Amazon EKS 最佳化 Windows AMI 管理

Amazon EKS 最佳化 AMI 是根據 Windows Server 2019 和 Windows Server 2022 所建置。這些 AMI 已設定為 Amazon EKS 節點的基礎映像。預設情況下，AMI 會包括以下組件：

- kubelet
- kube-proxy
- 適用於 Kubernetes 的 AWS IAM Authenticator
- csi-proxy
- containerd

您可以查詢 AWS Systems Manager 參數存放區 API，以程式設計方式擷取 Amazon EKS 最佳化 AMIs 的 Amazon Machine Image (AMI) ID。此參數讓您無需手動查詢 Amazon EKS 最佳化 AMI ID。如需 Systems Manager 參數存放區 API 的詳細資訊，請參閱 GetParameter。您的使用者帳戶必須具有 ssm：GetParameter IAM 許可，才能擷取 Amazon EKS 最佳化 AMI 中繼資料。

下列範例會擷取適用於 Windows Server 2019 LTSC Core 的最新 Amazon EKS 最佳化 AMI 的 AMI ID。AMI 名稱中列出的版本編號與其準備使用的對應 Kubernetes 組建相關。

```
aws ssm get-parameter --name /aws/service/ami-windows-latest/Windows_Server-2019-
English-Core-EKS_Optimized-1.21/image_id --region us-east-1 --query "Parameter.Value"
 --output text
```

輸出範例：

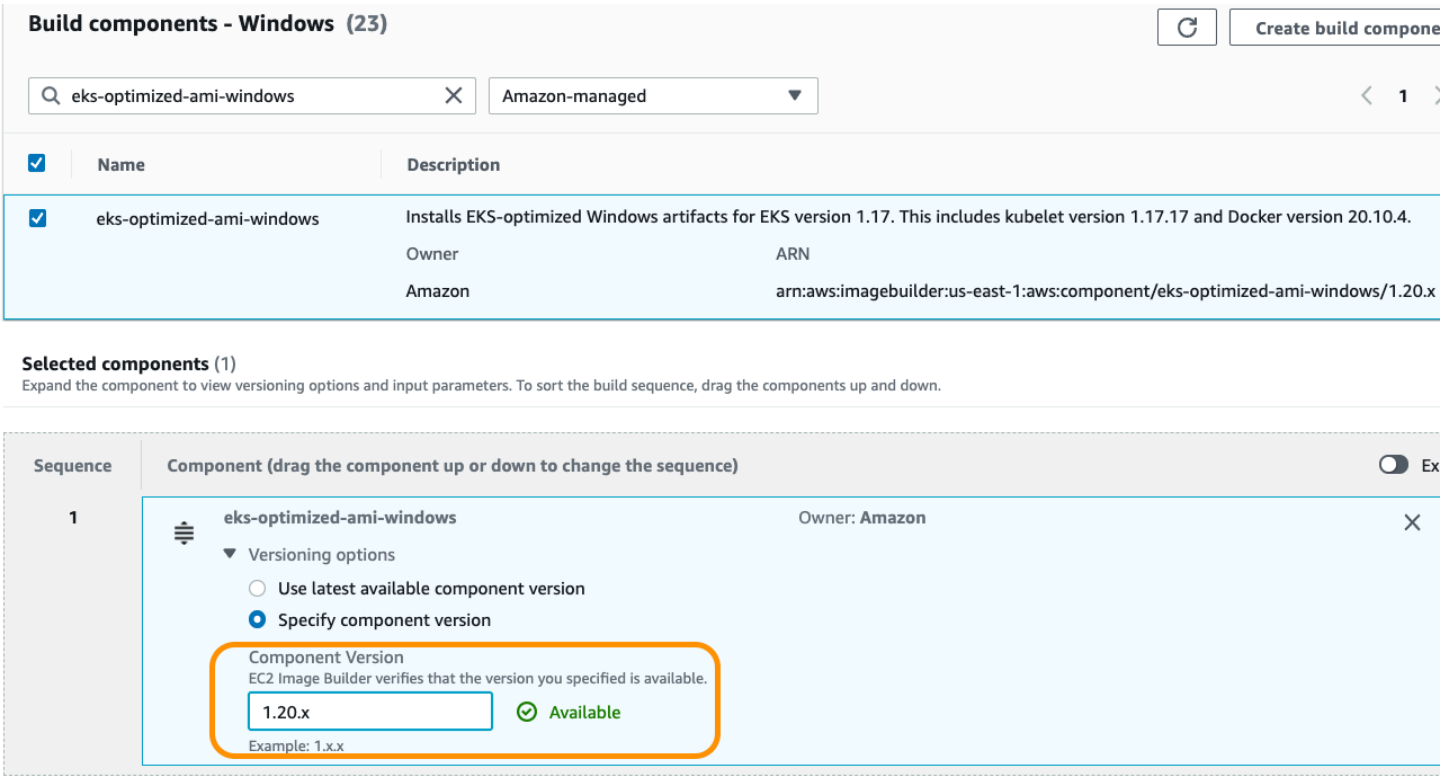
```
ami-09770b3eec4552d4e
```

管理您自己的 Amazon EKS 最佳化 Windows AMI

生產環境的重要步驟是在整個 Amazon EKS 叢集中維護相同的 Amazon EKS 最佳化 Windows AMI 和 kubelet 版本。

在 Amazon EKS 叢集中使用相同版本可減少故障診斷期間的時間，並提高叢集一致性。Amazon EC2 Image Builder 可協助建立和維護要在 Amazon EKS 叢集中使用的自訂 Amazon EKS 最佳化 Windows AMIs。

使用 Amazon EC2 Image Builder 在 Windows Server 版本、AWS Windows Server AMI 發行日期和/或作業系統建置版本之間進行選取。建置元件步驟可讓您在現有的 EKS Optimized Windows Artifacts 和 kubelet 版本之間進行選取。如需詳細資訊：https：//https://docs.aws.amazon.com/eks/latest/userguide/eks-custom-ami-windows.html



注意：選取基礎映像之前，請參閱 Windows Server 版本和授權一節，以取得有關發行頻道更新的重要詳細資訊。

為自訂 EKS 最佳化 AMIs 設定更快速的啟動

使用自訂 Windows Amazon EKS 最佳化 AMI 時，透過啟用快速啟動功能，Windows 工作者節點的啟動速度最多可加快 65%。此功能會維護一組預先佈建的快照，這些快照具有 Sysprep 專業、Windows Out of Box Experience (OOBE) 步驟，以及已完成的必要重新啟動。然後，這些快照會用於後續的啟動，減少擴展或取代節點的時間。只有您透過 EC2 主控台或 AWS CLI 擁有的 AMIs 才能啟用快速啟動，且可設定維護的快照數量。

注意：快速啟動與預設 Amazon 提供的 EKS 最佳化 AMI 不相容，請先如上述建立自訂 AMI，然後再嘗試啟用它。

如需詳細資訊：AWS Windows AMIs- 設定您的 AMI 以加快啟動速度

在自訂 AMIs上快取 Windows 基本層

Windows 容器映像大於其 Linux 對應映像。如果您執行任何容器化 .NET Framework 型應用程式，平均映像大小約為 8.24GB。在 Pod 排程期間，容器映像必須完全提取並擷取在磁碟中，Pod 才會達到執行中狀態。

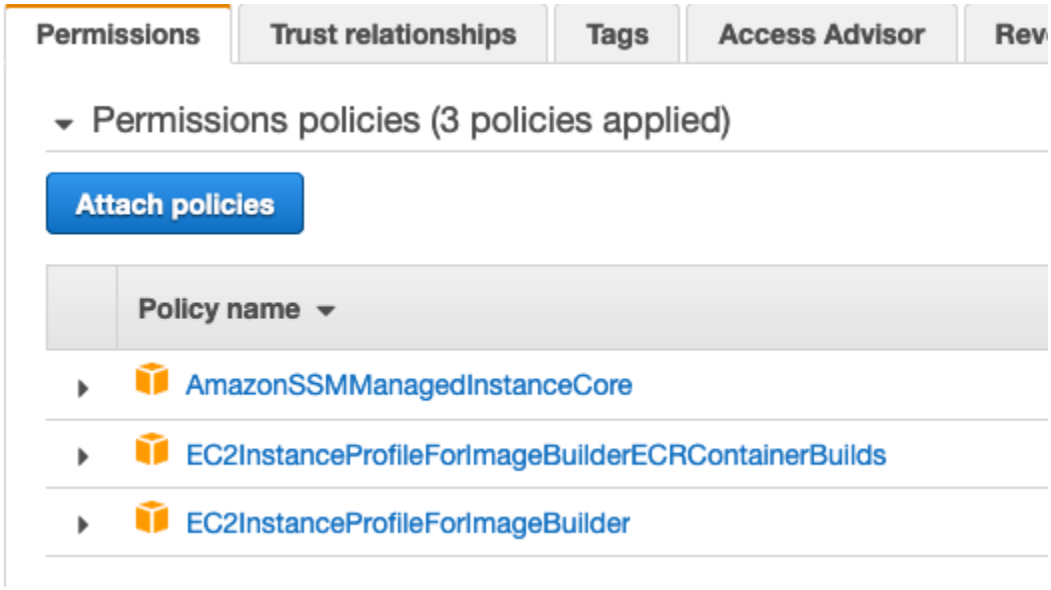
在此過程中，容器執行時間（容器）會在磁碟中提取和擷取整個容器映像。提取操作是平行程序，這表示容器執行時間會平行提取容器映像層。相反地，擷取操作會在循序程序中發生，而且是 I/O 密集型操作。因此，容器映像可能需要超過 8 分鐘才能完全解壓縮，並準備好供容器執行時間（容器）使用，因此，Pod 啟動時間可能需要幾分鐘。

如修補 Windows Server 和容器主題所述，您可以選擇使用 EKS 建置自訂 AMI。在 AMI 準備期間，您可以新增額外的 EC2 映像建置器元件，以在本機提取所有必要的 Windows 容器映像，然後產生 AMI。此策略將大幅減少 Pod 達到執行中狀態的時間。

在 Amazon EC2 Image Builder 上，建立元件以下載必要的映像，並將其連接至映像配方。下列範例會從 ECR 儲存庫提取特定映像。

```
name: ContainerdPull
description: This component pulls the necessary containers images for a cache strategy.
schemaVersion: 1.0

phases:
  - name: build
    steps:
      - name: containerdpull
        action: ExecutePowerShell
        inputs:
          commands:
            - Set-ExecutionPolicy Unrestricted -Force
            - (Get-ECRLoginCommand).Password | docker login --username AWS --password-
stdin 111000111000.dkr.ecr.us-east-1.amazonaws.com
            - ctr image pull mcr.microsoft.com/dotnet/framework/aspnet:latest
            - ctr image pull 111000111000.dkr.ecr.us-east-1.amazonaws.com/
myappcontainerimage:latest
```

為了確保下列元件如預期運作，請檢查 EC2 Image Builder (EC2InstanceProfileForImageBuilder) 所使用的 IAM 角色是否具有連接的政策：



部落格文章

在下列部落格文章中，您將逐步了解如何實作自訂 Amazon EKS Windows AMIs快取策略：

使用 EC2 映像建置器和映像快取策略加速 Windows 容器啟動時間

為 Windows Pod 和容器設定 gMSA

什麼是 gMSA 帳戶

Windows 型應用程式，例如 .NET 應用程式通常會使用 Active Directory 做為身分提供者，使用 NTLM 或 Kerberos 通訊協定提供授權/身分驗證。

與 Active Directory 交換 Kerberos 票證的應用程式伺服器需要加入網域。Windows 容器不支援網域聯結，而且由於容器是暫時性資源，對 Active Directory RID 集區造成負擔，因此沒有什麼意義。

不過，管理員可以利用 gMSA Active Directory 帳戶來交涉 Windows 容器、NLB 和伺服器陣列等資源的 Windows 身分驗證。

Windows 容器和 gMSA 使用案例

利用 Windows 身分驗證並以 Windows 容器執行的應用程式受益於 gMSA，因為 Windows 節點用於代表容器交換 Kerberos 票證。有兩個選項可用於設定 Windows 工作者節點以支援 gMSA 整合：

1 - 加入網域的 Windows 工作者節點

在此設定中，Windows 工作者節點在 Active Directory 網域中加入網域，Windows 工作者節點的 AD 電腦帳戶用於對 Active Directory 進行身分驗證，並擷取要與 Pod 搭配使用的 gMSA 身分。

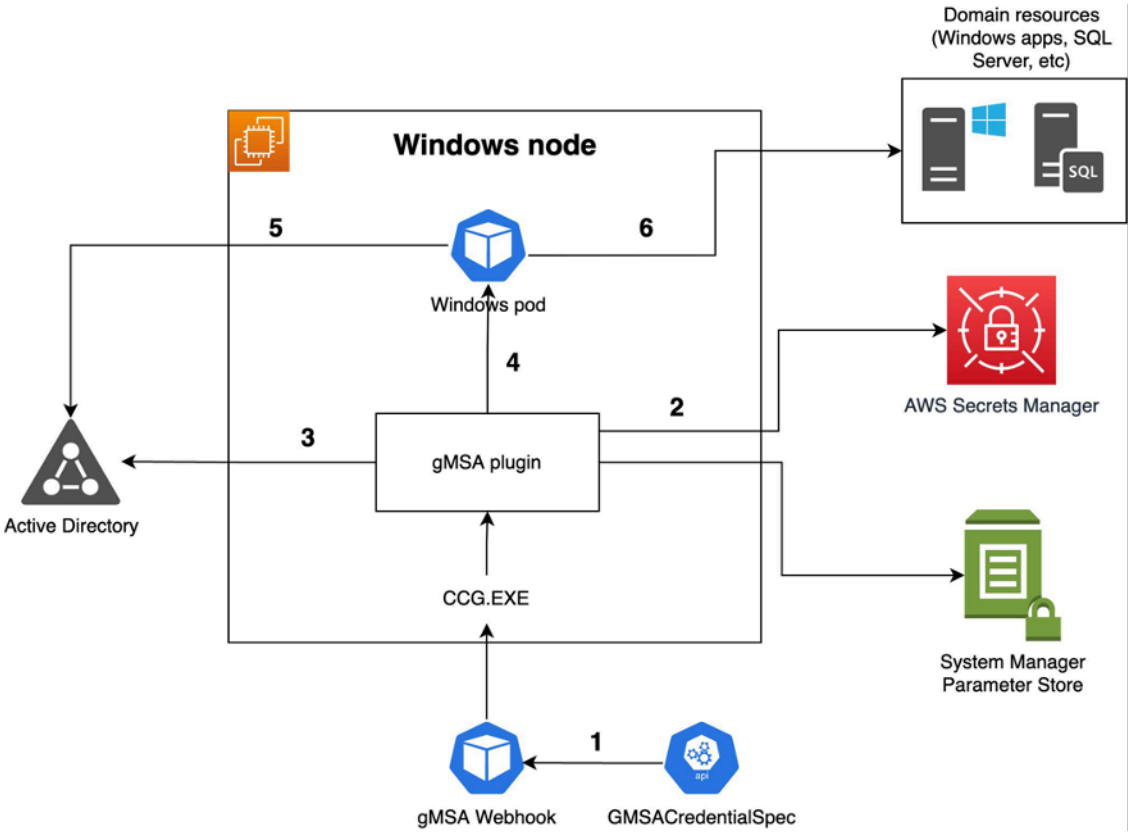
在加入網域的方法中，您可以使用現有的 Active Directory GPOs 輕鬆管理和強化 Windows 工作者節點；不過，在加入 Kubernetes 叢集的 Windows 工作者節點期間會產生額外的操作額外負荷和延遲，因為它需要在節點啟動期間進行額外的重新啟動，並在 Kubernetes 叢集終止節點後進行 Active Directory 車庫清理。

在下列部落格文章中，您將找到如何實作加入網域的 Windows 工作者節點方法的詳細step-by-step說明：

Amazon EKS Windows Pod 上的 Windows 身分驗證

2 - 無網域 Windows 工作者節點

在此設定中，Windows 工作者節點未加入 Active Directory 網域，而「可攜式」身分 （使用者/密碼）用於對 Active Directory 進行身分驗證，並擷取要與 Pod 搭配使用的 gMSA 身分。



可攜式身分是 Active Directory 使用者；身分 （使用者/密碼） 存放在 AWS Secrets Manager 或 AWS System Manager 參數存放區，而稱為 ccg_plugin 的 AWS 開發外掛程式將用於從 AWS Secrets

Manager 或 AWS System Manager 參數存放區擷取此身分，並將其傳遞至容器化以擷取 gMSA 身分，並使其可供 Pod 使用。

在此無網域方法中，您可以在使用 gMSA 時，在 Windows 工作者節點啟動期間不進行任何 Active Directory 互動，並降低 Active Directory 管理員的操作開銷。

在下列部落格文章中，您將找到如何實作無網域 Windows 工作者節點方法的詳細step-by-step說明：

Amazon EKS Windows Pod 的無網域 Windows 身分驗證

重要注意事項

雖然 Pod 能夠使用 gMSA 帳戶，但也必須相應地設定應用程式或服務以支援 Windows 身分驗證，例如，若要設定 Microsoft IIS 以支援 Windows 身分驗證，您應該透過 dockerfile 進行準備：

```
RUN Install-WindowsFeature -Name Web-Windows-Auth -IncludeAllSubFeature
RUN Import-Module WebAdministration; Set-ItemProperty 'IIS:\AppPools\SiteName' -name
 processModel.identityType -value 2
RUN Import-Module WebAdministration; Set-WebConfigurationProperty -Filter '/
system.webServer/security/authentication/anonymousAuthentication' -Name Enabled -Value
 False -PSPath 'IIS:\' -Location 'SiteName'
RUN Import-Module WebAdministration; Set-WebConfigurationProperty -Filter '/
system.webServer/security/authentication/windowsAuthentication' -Name Enabled -Value
 True -PSPath 'IIS:\' -Location 'SiteName'
```

Windows 工作者節點強化

作業系統強化是作業系統組態、修補和移除不必要軟體套件的組合，旨在鎖定系統並減少攻擊面。最佳實務是使用貴公司所需的強化組態來準備自己的 EKS Optimized Windows AMI。

AWS 每月提供新的 EKS 最佳化 Windows AMI，其中包含最新的 Windows Server 安全修補程式。不過，無論使用者是使用自我管理還是受管節點群組，使用者仍有責任透過套用必要的作業系統組態來強化其 AMI。

Microsoft 提供各種工具，例如 Microsoft 安全合規工具組和安全基準，可協助您根據您的安全政策需求實現強化。CIS 基準也可供使用，並且應該在生產環境的 Amazon EKS Optimized Windows AMI 上實作。

使用 Windows Server Core 減少攻擊面

Windows Server Core 是最小的安裝選項，可作為 EKS Optimized Windows AMI 的一部分使用。部署 Windows Server Core 有幾個優點。首先，其磁碟佔用空間相對較小，在具有桌面體驗的 Windows Server 上，在 Server Core 上為 6GB，在 Windows Server 上為 10GB。其次，它具有較小的攻擊面，因為它的程式碼基礎和可用的 APIs 較小。

AWS 每月為客戶提供新的 Amazon EKS Optimized Windows AMIs，包含最新的 Microsoft 安全性修補程式，無論 Amazon EKS 支援的版本為何。根據最佳實務，Windows 工作者節點必須取代為以最新 Amazon EKS 最佳化 AMI 為基礎的新節點。任何執行超過 45 天的節點，如果沒有更新或節點取代，則缺少安全最佳實務。

避免 RDP 連線

遠端桌面通訊協定 (RDP) 是由 Microsoft 開發的連線通訊協定，可提供使用者圖形界面，以透過網路連線至其他 Windows 電腦。

根據最佳實務，您應該將 Windows 工作者節點視為暫時性主機。這表示沒有管理連線、沒有更新，也沒有故障診斷。任何修改和更新都應實作為新的自訂 AMI，並以更新 Auto Scaling 群組取代。請參閱修補 Windows 伺服器和容器，以及 Amazon EKS 最佳化 Windows AMI 管理。

在部署期間，透過在 ssh 屬性上傳遞 false 值來停用 Windows 節點上的 RDP 連線，如以下範例所示：

```
nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
```

如果需要存取 Windows 節點，請使用 AWS System Manager Session Manager 透過 AWS 主控台和 SSM 代理程式建立安全的 PowerShell 工作階段。若要了解如何實作解決方案，請使用 AWS Systems Manager Session Manager 監看安全存取 Windows 執行個體

若要使用 System Manager Session Manager，必須將額外的 IAM 政策套用至用來啟動 Windows 工作者節點的 IAM 角色。以下是`eksctl`叢集資訊清單中指定 AmazonSSMManagedInstanceCore 的範例：

```
 nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
  iam:
    attachPolicyARNs:
      - arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy
      - arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy
      - arn:aws:iam::aws:policy/ElasticLoadBalancingFullAccess
      - arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly
      - arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore
```

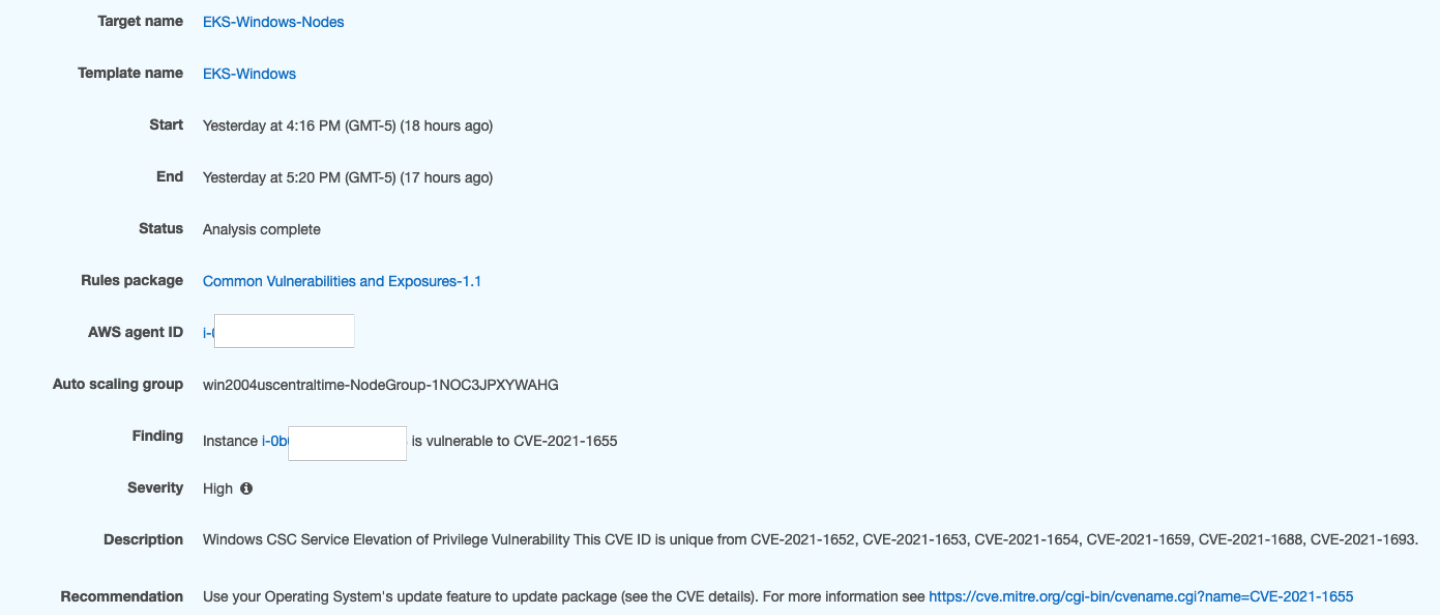
Amazon Inspector

> Amazon Inspector 是一種自動化安全評估服務，可協助改善部署在 AWS 上應用程式的安全性和合規性。Amazon Inspector 會自動評估應用程式是否有暴露、漏洞和與最佳實務的偏差。執行評估後，Amazon Inspector 會產生依嚴重性層級排定優先順序的安全調查結果詳細清單。這些調查結果可以直接檢閱，也可以作為詳細評估報告的一部分，這些報告可透過 Amazon Inspector 主控台或 API 取得。

Amazon Inspector 可用於在 Windows 工作者節點上執行 CIS Benchmark 評估，並可透過執行下列任務將其安裝在 Windows Server Core 上：

1. 下載下列 .exe 檔案：https：//https://inspector-agent.amazonaws.com/windows/installer/latest/AWSAgentInstall.exe
2. 將代理程式轉移到 Windows 工作者節點。
3. 在 PowerShell 上執行下列命令來安裝 Amazon Inspector Agent： `.\AWSAgentInstall.exe /install`

以下是第一次執行後的輸出。如您所見，它會根據 CVE 資料庫產生調查結果。您可以使用它來強化工作者節點，或根據強化的組態建立 AMI。



如需 Amazon Inspector 的詳細資訊，包括如何安裝 Amazon Inspector 代理程式、設定 CIS 基準評估和產生報告，請觀看使用 Amazon Inspector 影片改善 Windows 工作負載的安全性和合規性。

Amazon GuardDuty

> Amazon GuardDuty 是一種威脅偵測服務，可持續監控惡意活動和未經授權的行為，以保護存放在 Amazon S3 中的 AWS 帳戶、工作負載和資料。使用雲端，可簡化帳戶和網路活動的收集和彙總，但安全團隊可能需要耗費一些時間來持續分析事件日誌資料，以找出潛在的威脅。

透過使用 Amazon GuardDuty，您對 Windows 工作者節點的惡意動作具有可見性，例如 RDP 暴力破解和 Port Probe 攻擊。

使用 Amazon GuardDuty 影片觀看 Windows 工作負載的威脅偵測，了解如何在最佳化 EKS Windows AMI 上實作和執行 CIS 基準

Amazon EC2 for Windows 中的安全性

閱讀 Amazon EC2 Windows 執行個體的安全最佳實務，在每一層實作安全控制。

容器映像掃描

映像掃描是一種自動化漏洞評估功能，可透過掃描各種作業系統漏洞，協助改善應用程式容器映像的安全性。

目前，Amazon Elastic Container Registry (ECR) 只能掃描 Linux 容器映像是否有漏洞。不過，有第三方工具可以與現有的 CI/CD 管道整合，用於 Windows 容器映像掃描。

- Anchore
- PaloAlto Prisma 雲端
- Trend Micro - Deep Security Smart Check

若要進一步了解如何將這些解決方案與 Amazon Elastic Container Repository (ECR) 整合，請檢查：

- Anchore，在 Amazon Elastic Container Registry (ECR) 上掃描映像
- PaloAlto，在 Amazon Elastic Container Registry (ECR) 上掃描映像
- TrendMicro，在 Amazon Elastic Container Registry (ECR) 上掃描映像

Windows Server 版本和授權

Windows Server 版本

Amazon EKS Optimized Windows AMI 是以長期服務管道 (LTSC) 上的 Windows Server 2019 和 2022 Datacenter 版本為基礎。Datacenter 版本對工作者節點上執行的容器數量沒有限制。如需詳細資訊：https：//https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/faq

長期服務管道 (LTSC)

先前稱為「長期服務分支」，這是您已熟悉的發行模型，其中每 2-3 年會發行新的 Windows Server 主要版本。使用者有權獲得 5 年的主流支援和 5 年延長支援。

授權

使用 Windows Server 型 AMI 啟動 Amazon EC2 執行個體時，Amazon 會為您支付授權成本和授權合規。

記錄

容器化應用程式通常會將應用程式日誌導向 STDOUT。容器執行期會捕捉這些日誌，並對其執行一些操作 - 通常寫入檔案。這些檔案的儲存位置取決於容器執行時間和組態。

與 Windows Pod 的一個基本差異是它們不會產生 STDOUT。您可以執行 LogMonitor，從執行 Windows 容器和管道格式化日誌輸出擷取 ETW （適用於 Windows 的事件追蹤）、Windows 事件日

誌和其他應用程式特定日誌到 STDOUT。然後，您可以使用 fluent-bit 或 fluentd 將這些日誌串流到所需的目的地，例如 Amazon CloudWatch。

日誌收集機制會從 Kubernetes Pod 擷取 STDOUT/STDERR 日誌。DaemonSet 是從容器收集日誌的常見方式。它可讓您獨立於應用程式管理日誌routing/filtering/enrichment。流利的 DaemonSet 可用來將這些日誌和任何其他應用程式產生的日誌串流到所需的日誌彙總器。

此處說明從 Windows 工作負載到 CloudWatch 的日誌串流詳細資訊 https://aws.amazon.com/blogs/containers/streaming-logs-from-amazon-eks-windows-pods-to-amazon-cloudwatch-logs-using-fluentd/

記錄建議

在 Kubernetes 中操作 Windows 工作負載時，一般記錄最佳實務並無不同。

- 一律記錄結構化日誌項目 (JSON/SYSLOG)，讓處理日誌項目變得更輕鬆，因為這類結構化格式有許多預先編寫的剖析器。
- 集中日誌 - 專用日誌容器可用於收集日誌訊息，並將日誌訊息從所有容器轉送至目的地
- 保持日誌詳細程度，除除除除除偵錯之外。Verbosity 會對記錄基礎設施造成許多壓力，而且在噪音中可能會遺失重大事件。
- 一律記錄應用程式資訊以及交易/請求 ID 以供追蹤。Kubernetes 物件不會攜帶應用程式名稱，因此例如，偵錯日誌時，Pod 名稱`windows-twryrqyw`可能沒有任何意義。這有助於使用彙總日誌對應用程式進行可追蹤性和故障診斷。

  如何產生這些交易/關聯 ID 的方式取決於程式設計建構。但是，一種非常常見的模式是使用記錄 Aspect/攔截器，它可以使用 MDC （映射診斷內容）將唯一的交易/關聯 ID 注入到每個傳入的請求，如下所示：

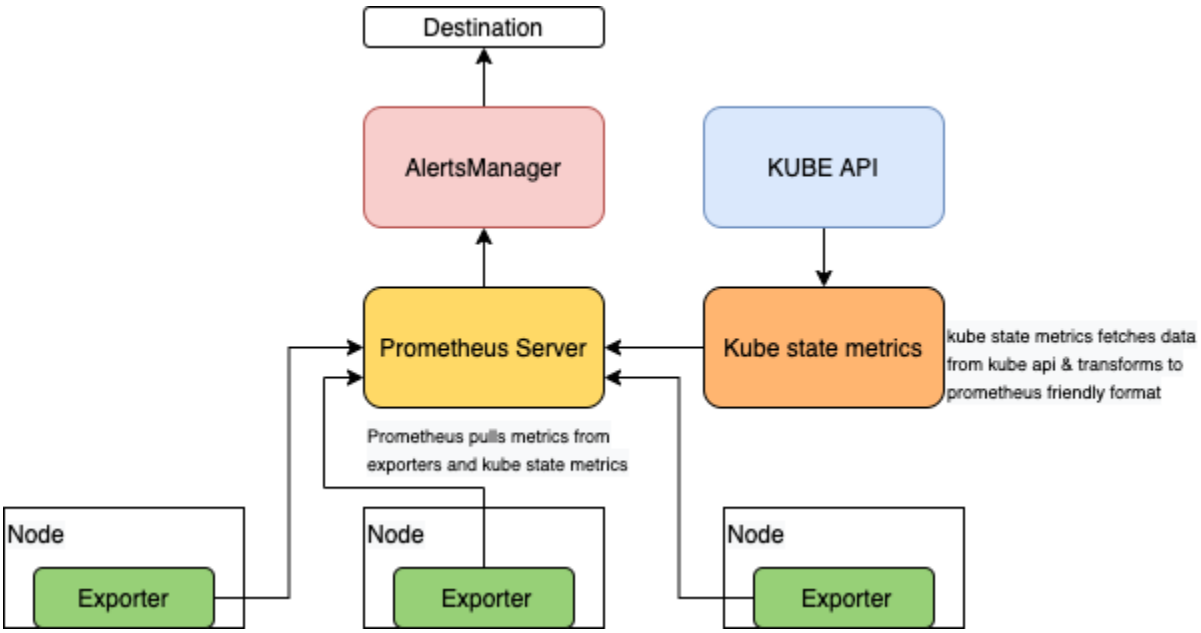
```
import org.slf4j.MDC;
import java.util.UUID;
Class LoggingAspect { //interceptor

    @Before(value = "execution(* *.*(..))")
    func before(...) {
        transactionId = generateTransactionId();
        MDC.put(CORRELATION_ID, transactionId);
    }

    func generateTransactionId() {
```

```
        return UUID.randomUUID().toString();
    }
}
```

監控

Prometheus，一個漸進式 CNCF 專案是迄今為止最熱門的監控系統，具有與 Kubernetes 的原生整合。Prometheus 會收集容器、Pod、節點和叢集的指標。此外，Prometheus 利用 AlertsManager，可讓您編寫警示程式，在叢集發生錯誤時警告您。Prometheus 會將指標資料儲存為由指標名稱和索引鍵/值對識別的時間序列資料。Prometheus 包含離開使用稱為 PromQL 的語言進行查詢，Prometheus 查詢語言很短。

Prometheus 指標集合的高階架構如下所示：



Prometheus 使用提取機制，並使用匯出工具從目標抓取指標，並使用 kube 狀態指標從 Kubernetes API 抓取指標。這表示應用程式和服務必須公開包含 Prometheus 格式指標的 HTTP(S) 端點。然後，Prometheus 會根據其組態，定期從這些 HTTP(S) 端點提取指標。

匯出工具可讓您使用第三方指標做為 Prometheus 格式指標。Prometheus 匯出工具通常部署在每個節點上。如需匯出者的完整清單，請參閱 Prometheus 匯出者。雖然節點匯出器適用於匯出 Linux 節點的主機硬體和作業系統指標，但不適用於 Windows 節點。

當您使用穩定的 Prometheus Helm Chart 時，在具有 Windows 節點的混合節點 EKS 叢集中，您會在 Windows 節點上看到失敗的 Pod，因為此匯出工具不適用於 Windows。您需要分開處理 Windows 工作者集區，而是在 Windows 工作者節點群組上安裝 Windows 匯出工具。

若要設定 Windows 節點的 Prometheus 監控，您需要在 Windows 伺服器本身下載並安裝 WMI 匯出工具，然後在 Prometheus 組態檔案的湊集組態內設定目標。版本頁面提供所有可用的 .msi 安裝程式，以及個別的功能集和錯誤修正。安裝程式會將 windows_exporter 設定為 Windows 服務，並在 Windows 防火牆中建立例外狀況。如果安裝程式在沒有任何參數的情況下執行，匯出程式將以已啟用收集器、連接埠等的預設設定執行。

您可以查看本指南的排程最佳實務章節，其中建議使用污點/容錯或 RuntimeClass 來選擇性地將節點匯出器僅部署到 linux 節點，而 Windows 匯出器會在啟動節點時安裝在 Windows 節點上，或使用您選擇的組態管理工具（例如 Chef、Ansible、SSM 等）。

請注意，與節點匯出器安裝為協助程式集 的 linux 節點不同，在 Windows 節點上，WMI 匯出器會安裝在主機本身。匯出者將匯出指標，例如 CPU 用量、記憶體和磁碟 I/O 用量，也可以用來監控 IIS 網站和應用程式、網路介面和服務。

Windows_exporter 預設會公開已啟用收集器的所有指標。這是收集指標以避免錯誤的推薦方法。不過，對於進階使用，Windows_exporter 可以傳遞選用的收集器清單來篩選指標。Prometheus 組態中的 collect【】 參數可讓您執行此操作。

Windows 的預設安裝步驟包括在引導程序期間使用 引數下載並啟動匯出工具做為服務，例如您要篩選的收集器。

```
> Powershell Invoke-WebRequest https://github.com/prometheus-community/
windows_exporter/releases/download/v0.13.0/windows_exporter-0.13.0-amd64.msi -OutFile
 <DOWNLOADPATH>

> msiexec /i <DOWNLOADPATH>
 ENABLED_COLLECTORS="cpu,cs,logical_disk,net,os,system,container,memory"
```

根據預設，指標可以在連接埠 9182 上的 /metrics 端點上抓取。此時，Prometheus 可以透過將下列 scrape_config 新增至 Prometheus 組態來取用指標

```
scrape_configs:
    - job_name: "prometheus"
      static_configs:
        - targets: ['localhost:9090']
    ...
    - job_name: "wmi_exporter"
      scrape_interval: 10s
      static_configs:
        - targets: ['<windows-node1-ip>:9182', '<windows-node2-ip>:9182', ...]
```

Prometheus 組態是使用 重新載入

```
> ps aux | grep prometheus
> kill HUP <PID>
```

新增目標的更好建議方式是使用稱為 ServiceMonitor 的自訂資源定義，該定義是做為 Prometheus 運算子的一部分】，提供 ServiceMonitor 物件的定義，以及可啟用 ServiceMonitors 我們定義並自動建置必要 Prometheus 組態的控制器。

ServiceMonitor 會宣告指定應如何監控 Kubernetes 服務群組，以定義您希望從 Kubernetes 中抓取指標的應用程式。在 ServiceMonitor 中，我們指定運算子可用來識別 Kubernetes 服務的 Kubernetes 標籤，而 Kubernetes 服務又會識別我們想要監控的 Pod。

為了利用 ServiceMonitor，請為 Windows 節點建立指向特定 Windows 目標的端點物件、無周邊服務和 ServiceMontor。

```
apiVersion: v1
kind: Endpoints
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: kube-system
subsets:
- addresses:
  - ip: NODE-ONE-IP
    targetRef:
      kind: Node
      name: NODE-ONE-NAME
  - ip: NODE-TWO-IP
    targetRef:
      kind: Node
      name: NODE-TWO-NAME
  - ip: NODE-THREE-IP
    targetRef:
      kind: Node
      name: NODE-THREE-NAME
  ports:
  - name: http-metrics
    port: 9182
    protocol: TCP
```

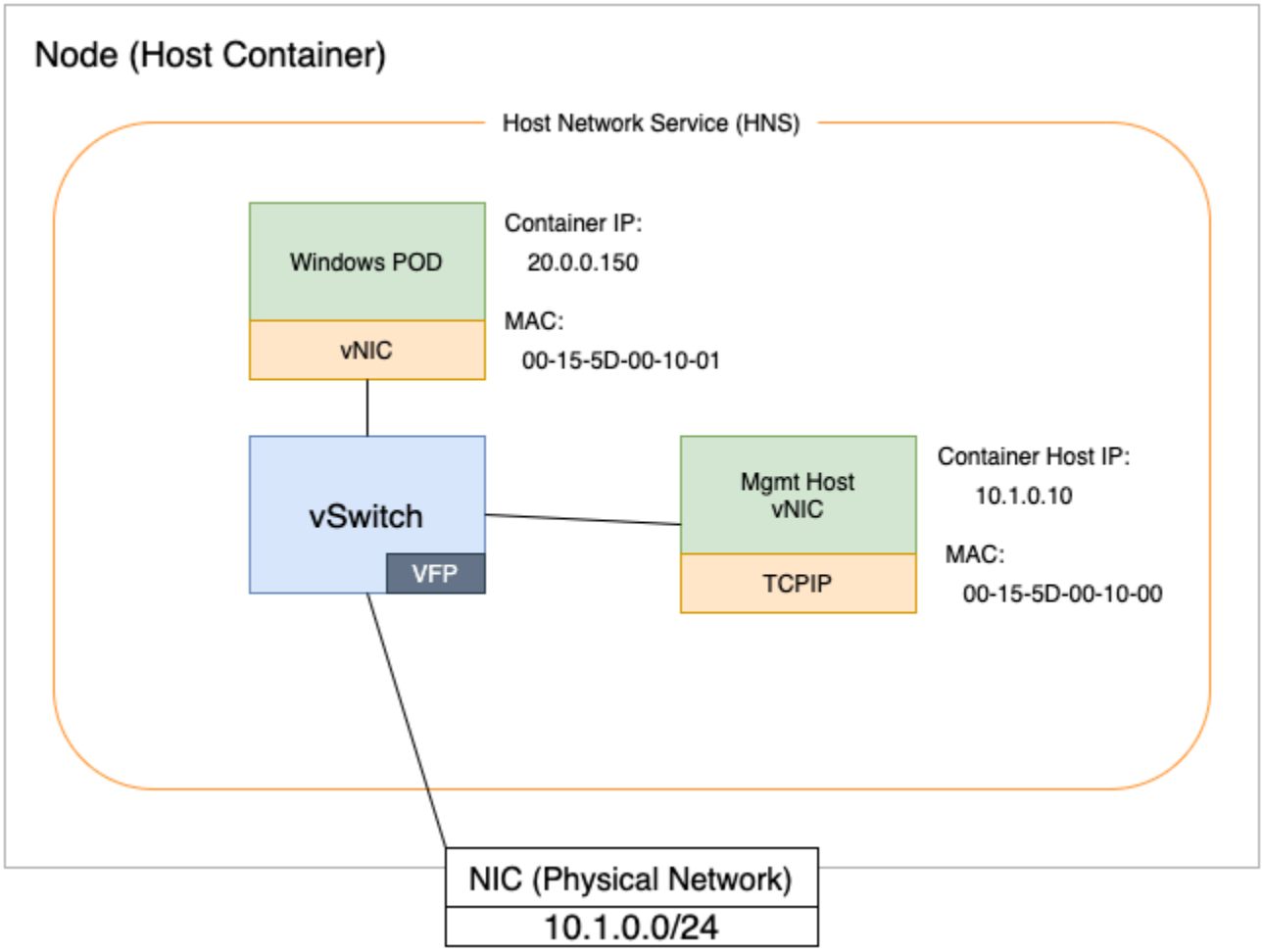
```
---
apiVersion: v1
kind: Service ##Headless Service
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: kube-system
spec:
  clusterIP: None
  ports:
  - name: http-metrics
    port: 9182
    protocol: TCP
    targetPort: 9182
  sessionAffinity: None
  type: ClusterIP

---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor ##Custom ServiceMonitor Object
metadata:
  labels:
    k8s-app: wmiexporter
  name: wmiexporter
  namespace: monitoring
spec:
  endpoints:
  - interval: 30s
    port: http-metrics
  jobLabel: k8s-app
  namespaceSelector:
    matchNames:
    - kube-system
  selector:
    matchLabels:
      k8s-app: wmiexporter
```

如需 運算子和 ServiceMonitor 使用方式的詳細資訊，請參閱官方運算子文件。請注意，Prometheus 確實支援使用許多服務探索選項的動態目標探索。

Windows 網路

Windows 容器聯網概觀

Windows 容器與 Linux 容器基本上不同。Linux 容器使用 Linux 建構，例如命名空間、聯集檔案系統和 cgroup。在 Windows 上，這些建構是由主機運算服務 (HCS) 從容器擷取。HCS 充當位於 Windows 上容器實作上方的 API 層。Windows 容器也會利用 Host Network Service (HNS) 來定義節點上的網路拓撲。



從聯網的角度來看，HCS 和 HNS 可讓 Windows 容器像虛擬機器一樣運作。例如，每個容器都有連接到 Hyper-V 虛擬切換 (vSwitch) 的虛擬網路轉接器 (vNIC)，如上圖所示。

IP 地址管理

Amazon EKS 中的節點會使用其彈性網路界面 (ENI) 來連線至 AWS VPC 網路。目前，每個 Windows 工作者節點僅支援單一 ENI。Windows 節點的 IP 地址管理是由在控制平面中執行的 VPC Resource Controller 執行。如需 Windows 節點 IP 地址管理工作流程的詳細資訊，請參閱此處。

Windows 工作者節點可支援的 Pod 數量取決於節點的大小和可用的 IPv4 地址數量。您可以計算節點上可用的 IPv4 地址，如下所示：

- 根據預設，只有次要 IPv4 地址會指派給 ENI。在這種情況下：

```
Total IPv4 addresses available for Pods = Number of supported IPv4 addresses in the
 primary interface - 1
```

  我們從總數中減去一個，因為一個 IPv4 地址將用作 ENI 的主要地址，因此無法配置給 Pod。

- 如果叢集已透過啟用字首委派功能設定為高 Pod 密度，則

```
Total IPv4 addresses available for Pods = (Number of supported IPv4 addresses in the
 primary interface - 1) * 16
```

  在這裡，VPC 資源控制器將配置 ，而不是配置次要 IPv4 地址，`/28 prefixes`因此，可用 IPv4 地址的整體數量將增加 16 次。

使用上述公式，我們可以計算以 m5.large 執行個體為基礎的 Windows 工作者節點 Pod 上限，如下所示：

- 根據預設，在次要 IP 模式下執行時 -

```
10 secondary IPv4 addresses per ENI - 1 = 9 available IPv4 addresses
```

- 使用 時 `prefix delegation`-

```
(10 secondary IPv4 addresses per ENI - 1) * 16 = 144 available IPv4 addresses
```

如需執行個體類型可支援多少 IP 地址的詳細資訊，請參閱每個執行個體類型每個網路界面的 IP 地址。

另一個關鍵考量因素是網路流量的流程。使用 Windows 時，超過 100 個服務的節點會有連接埠耗盡的風險。當出現此條件時，節點會開始擲回錯誤，並顯示下列訊息：

「政策建立失敗：hcnCreateLoadBalancer 在 Win32 中失敗：指定的連接埠已存在。」

為了解決此問題，我們利用 Direct Server Return (DSR)。DSR 是非對稱網路負載分佈的實作。換句話說，請求和回應流量會使用不同的網路路徑。此功能可加速 Pod 之間的通訊，並降低連接埠耗盡的風險。因此，我們建議您在 Windows 節點上啟用 DSR。

預設會在 Windows Server SAC EKS 最佳化 AMIs 中啟用 DSR。對於 Windows Server 2019 LTSC EKS 最佳化 AMIs，您將需要使用以下指令碼和使用 Windows Server 2019 Full 或 Core 做為 `eksctl` nodeGroup 中的 amiFamily，在執行個體佈建期間啟用它。如需其他資訊，請參閱 eksctl 自訂 AMI。

```
nodeGroups:
- name: windows-ng
  instanceType: c5.xlarge
  minSize: 1
  volumeSize: 50
  amiFamily: WindowsServer2019CoreContainer
  ssh:
    allow: false
```

若要在 Windows Server 2019 及更高版本中使用 DSR，您需要在執行個體啟動期間指定下列 kube-proxy 旗標。您可以透過調整與自我管理節點群組啟動範本相關聯的使用者資料指令碼來執行此操作。

```
<powershell>
[string]$EKSBinDir = "$env:ProgramFiles\Amazon\EKS"
[string]$EKSBootstrapScriptName = 'Start-EKSBootstrap.ps1'
[string]$EKSBootstrapScriptFile = "$EKSBinDir\$EKSBootstrapScriptName"
(Get-Content $EKSBootstrapScriptFile).replace('"--proxy-mode=kernelspace",', '"--proxy-
mode=kernelspace", "--feature-gates WinDSR=true", "--enable-dsr",') | Set-Content
 $EKSBootstrapScriptFile
& $EKSBootstrapScriptFile -EKSClusterName "eks-windows" -APIServerEndpoint "https://
<REPLACE-EKS-CLUSTER-CONFIG-API-SERVER>" -Base64ClusterCA "<REPLACE-EKSCLUSTER-
CONFIG-DETAILS-CA>" -DNSClusterIP "172.20.0.10" -KubeletExtraArgs "--node-
labels=alpha.eksctl.io/cluster-name=eks-windows,alpha.eksctl.io/nodegroup-name=windows-
ng-ltsc2019 --register-with-taints=" 3>&1 4>&1 5>&1 6>&1
</powershell>
```

DSR 啟用可以依照 Microsoft 網路部落格和 AWS Lab 上的 Windows 容器中的指示進行驗證。

```
PS C:\temp> (Get-Content C:\temp\pmsx1eau.0ia\policy.txt | ConvertFrom-JSON).Policies


ExternalPort : 53
InternalPort : 53
IsDSR        : True
Protocol     : 17
Type         : ELB
VIPs         : {172.20.0.10}

ExternalPort : 53
InternalPort : 53
IsDSR        : True
Protocol     : 6
Type         : ELB
VIPs         : {172.20.0.10}

ExternalPort : 443
InternalPort : 443
IsDSR        : True
Protocol     : 6
Type         : ELB
VIPs         : {172.20.0.1}
```

如果保留可用的 IPv4 地址並盡量減少浪費對您的子網路至關重要，通常建議避免使用字首委派模式，如 Windows 字首模式中所述 - 何時避免。如果仍然需要使用字首委派，您可以採取步驟來最佳化子網路中的 IPv4 地址使用率。如需如何微調 IPv4 地址請求和配置程序的詳細說明，請參閱設定字首委派的參數。調整這些組態可協助您在保留 IPv4 地址和字首委派的 Pod 密度優勢之間取得平衡。

使用指派次要 IPv4 地址的預設設定時，目前沒有支援的組態可操作 VPC 資源控制器請求和配置 IPv4 地址的方式。更具體地說，`minimum-ip-target`和 `warm-ip-target` 僅支援字首委派模式。另請注意，在次要 IP 模式中，根據界面上的可用 IP 地址，VPC 資源控制器通常會代表您在節點上配置 3 個未使用的 IPv4 地址，以維持暖 IPs以加快 Pod 啟動時間。如果您想要將未使用的暖 IP 地址 IP 浪費降至最低，您可以瞄準在指定的 Windows 節點上排程更多 Pod，以便盡可能使用 ENI 的 IP 地址容量。更明確地說， IPs 如果節點和執行中的 Pod 已在使用 ENI 上的所有 IP 地址，您可以避免使用暖未使用的 IP。另一個解決方法可協助您解決子網路中 IP 地址可用性的限制，可能是探索增加子網路大小，或將您的 Windows 節點分成自己的專用子網路。

此外，請務必注意 Windows 節點目前不支援 IPv6。

容器網路界面 (CNI) 選項

AWSVPC CNI 是適用於 Windows 和 Linux 工作者節點的事實 CNI 外掛程式。雖然 AWSVPC CNI 滿足許多客戶的需求，但有時候您可能需要考慮替代方法，例如覆蓋網路，以避免 IP 耗盡。在這些情況下，可以使用 Calico CNI 取代 AWSVPC CNI。Project Calico 是由 Tigera 開發的開放原始碼軟體。該

軟體包含可與 EKS 搭配使用的 CNI。您可以在專案 Calico EKS 安裝頁面上找到在 EKS 中安裝 Calico CNI 的說明。

網路政策

最佳實務是從 Kubernetes 叢集上 Pod 之間的預設開放通訊模式變更為根據網路政策限制存取。開放原始碼 Project Calico 對同時使用 Linux 和 Windows 節點的網路政策具有強大的支援。此功能是獨立的，不依賴使用 Calico CNI。因此，我們建議您安裝 Calico 並將其用於網路政策管理。

如需在 EKS 中安裝 Calico 的說明，請參閱在 Amazon EKS 上安裝 Calico 頁面。

此外，Amazon EKS 安全最佳實務指南 - 網路區段中提供的建議同樣適用於具有 Windows 工作者節點的 EKS 叢集，不過，Windows 目前不支援某些功能，例如「Pod 安全群組」。

避免 OOM 錯誤

Windows 沒有與 Linux out-of-memory程序刪除程式。Windows 一律會將所有使用者模式記憶體配置視為虛擬，且 pagefile 是強制性的。淨效果是 Windows 不會像 Linux 那樣脫離記憶體條件。程序將分頁到磁碟，而不是因記憶體不足 (OOM) 而終止。如果記憶體過度佈建且所有實體記憶體耗盡，分頁可能會降低效能。

保留系統和 kubelet 記憶體

與 Linux 不同，Linux 會`--kubelet-reserve`擷取 kubelet、容器執行時間等 kubernetes 系統常駐程式的資源保留，以及`--system-reserve`擷取 sshd、udev 等作業系統常駐程式的資源保留。在 Windows 上，這些旗標不會擷取和設定在節點上執行的 kubelet 或程序的記憶體限制。

不過，您可以結合這些旗標來管理 NodeAllocatable，以使用 Pod 資訊清單記憶體資源限制來減少節點上的容量，以控制每個 Pod 的記憶體配置。使用此策略，您可以更好地控制記憶體配置，以及將 Windows 節點上的out-of-memory(OOM) 降至最低的機制。

在 Windows 節點上，最佳實務是為作業系統和程序保留至少 2GB 的記憶體。使用 `--kubelet-reserve`和/或 `--system-reserve` 減少 NodeAllocatable。

遵循 Amazon EKS 自我管理 Windows 節點文件，使用 CloudFormation 範本啟動具有 kubelet 組態自訂功能的新 Windows 節點群組。CloudFormation 有一個名為 的元素`BootstrapArguments`，與 相同`KubeletExtraArgs`。搭配下列旗標和值使用：

```
--kube-reserved memory=0.5Gi,ephemeral-storage=1Gi --system-
reserved memory=1.5Gi,ephemeral-storage=1Gi --eviction-hard
 memory.available<200Mi,nodefs.available<10%"
```

如果 eksctl 是部署工具，請檢查下列文件以自訂 kubelet 組態 https：//https://eksctl.io/usage/customizing-the-kubelet/

Windows 容器記憶體需求

根據 Microsoft 文件，NANO 的 Windows Server 基礎映像至少需要 30MB，而 Server Core 則需要 45MB。這些數字會隨著您新增 Windows 元件而增加，例如 .NET Framework、Web Services as IIS 和應用程式。

您必須了解 Windows 容器映像所需的最低記憶體量，即基礎映像及其應用程式層，並將其設定為 Pod 規格中的容器資源/請求。您也應該設定限制，以避免 Pod 在應用程式發生問題時消耗所有可用的節點記憶體。

在下列範例中，當 Kubernetes 排程器嘗試將 Pod 放置在節點上時，會使用 Pod 的請求來判斷哪個節點有足夠的資源可供排程。

```
spec:
 - name: iis
   image: mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
   resources:
     limits:
       cpu: 1
       memory: 800Mi
     requests:
       cpu: .1
       memory: 128Mi
```

結論

使用此方法可將記憶體耗盡的風險降至最低，但無法防止發生這種情況。使用 Amazon CloudWatch 指標，您可以在記憶體耗盡時設定提醒和修復。

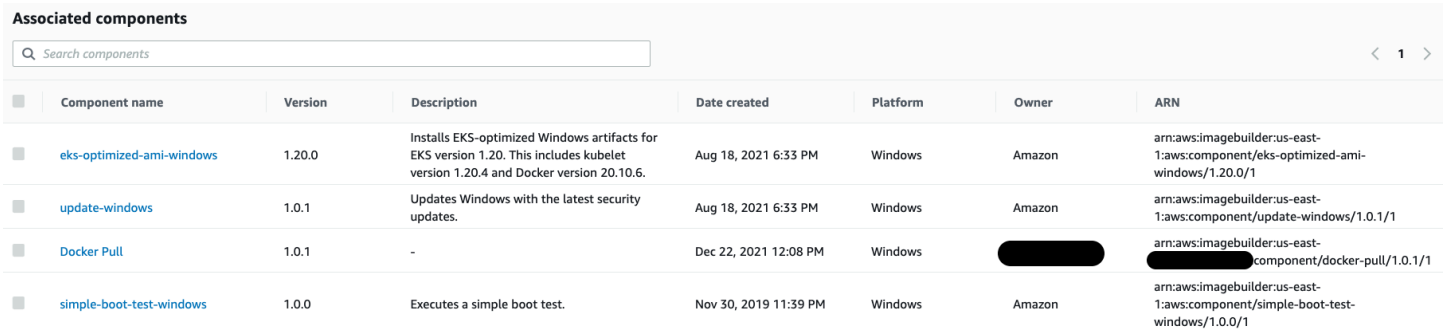
修補 Windows 伺服器和容器

修補 Windows Server 是 Windows 管理員的標準管理任務。這可以使用 Amazon System Manager - Patch Manager、WSUS、System Center Configuration Manager 等不同工具來完成。不過，Amazon

EKS 叢集中的 Windows 節點不應視為一般 Windows 伺服器。它們應視為不可變伺服器。簡單來說，避免更新現有的節點，只要根據新的更新 AMI 啟動新的節點即可。

使用 EC2 Image Builder，您可以透過建立配方和新增元件來自動化 AMIs 建置。

下列範例顯示元件，這些元件可以是由 AWS (Amazon 受管） 建置的預先存在元件，以及您建立的元件 （由我擁有）。請密切注意稱為 update-windows 的 Amazon 受管元件，這會在透過 EC2 Image Builder 管道產生 AMI 之前更新 Windows Server。



Associated components

Search components

1

Component name	Version	Description	Date created	Platform	Owner	ARN
eks-optimized-ami-windows	1.20.0	Installs EKS-optimized Windows artifacts for EKS version 1.20. This includes kubelet version 1.20.4 and Docker version 20.10.6.	Aug 18, 2021 6:33 PM	Windows	Amazon	arn:aws:imagebuilder:us-east-1:aws:component/eks-optimized-ami-windows/1.20.0/1
update-windows	1.0.1	Updates Windows with the latest security updates.	Aug 18, 2021 6:33 PM	Windows	Amazon	arn:aws:imagebuilder:us-east-1:aws:component/update-windows/1.0.1/1
Docker Pull	1.0.1	-	Dec 22, 2021 12:08 PM	Windows	[redacted]	arn:aws:imagebuilder:us-east-[redacted]component/docker-pull/1.0.1/1
simple-boot-test-windows	1.0.0	Executes a simple boot test.	Nov 30, 2019 11:39 PM	Windows	Amazon	arn:aws:imagebuilder:us-east-1:aws:component/simple-boot-test-windows/1.0.0/1

EC2 Image Builder 可讓您根據 Amazon Managed Public AMI 建置 AMIs，並自訂它們以符合您的業務需求。然後，您可以將這些 AMIs 與啟動範本建立關聯，以便將新的 AMI 連結至 EKS 節點群組建立的 Auto Scaling 群組。完成後，您可以開始終止現有的 Windows 節點，並根據新的更新 AMI 啟動新的節點。

推送和提取 Windows 映像

Amazon 發佈 EKS 最佳化 AMIs，其中包含兩個快取的 Windows 容器映像。

```
mcr.microsoft.com/windows/servercore
mcr.microsoft.com/windows/nanoserver
```

```
REPOSITORY                                                      TAG                      IMAGE ID       CREATED          SIZE
mcr.microsoft.com/windows/servercore                            ltsc2019                 617dd5ead5b8   6 weeks ago      5.74GB
mcr.microsoft.com/windows/nanoserver                            1809                     8334af2fa7ba   6 weeks ago      257MB
[redacted].dkr.ecr.us-east-1.amazonaws.com/fluentd-windows-coreltsc   latest                   721afca2c725   7 weeks ago      6.96GB
fluent/fluentd                                                  v1.14-windows-ltsc2019-1 721afca2c725   7 weeks ago      6.96GB
amazonaws.com/eks/pause-windows                                 latest                   6392f69ae6e7   10 months ago    255MB
```

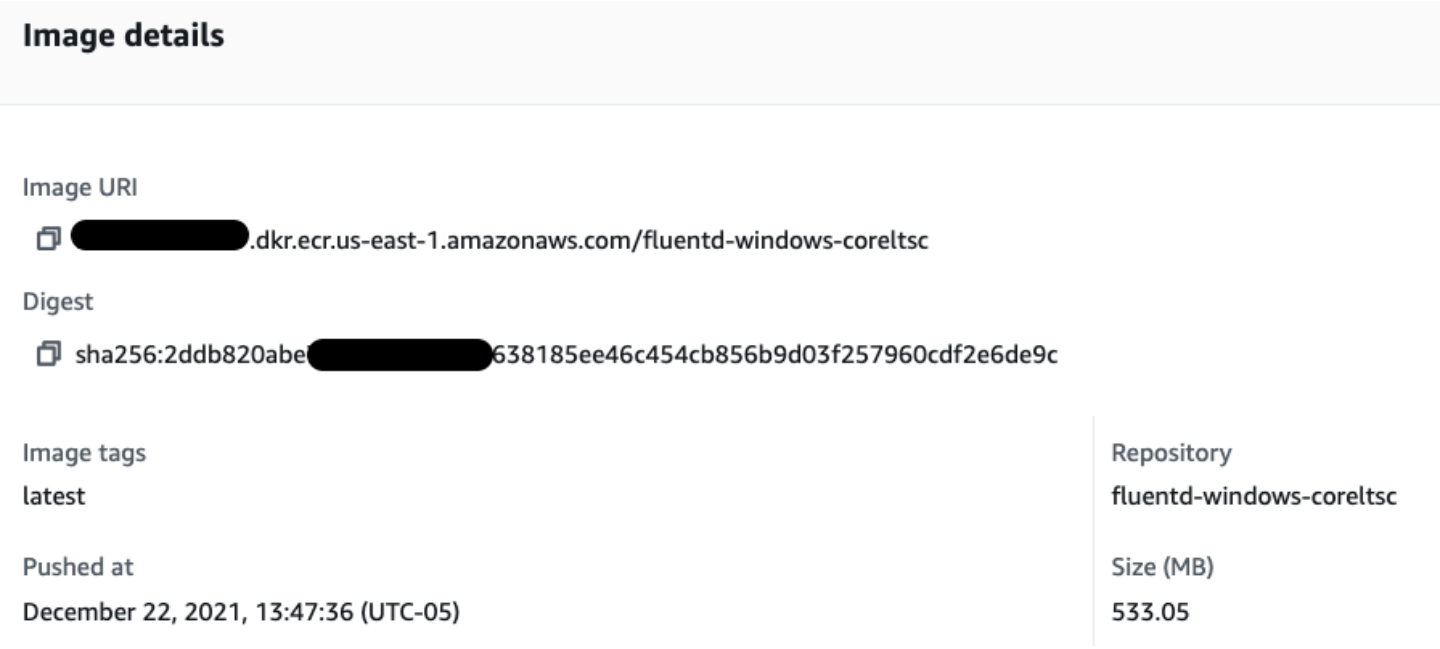
快取映像會在主要作業系統上的更新之後更新。當 Microsoft 發行直接影響 Windows 容器基礎映像的新 Windows 更新時，更新將作為主要作業系統上的一般 Windows Update 啟動。讓環境保持在up-to-date可在節點和容器層級提供更安全的環境。

Windows 容器映像的大小會影響推送/提取操作，這可能會導致容器啟動時間變慢。快取 Windows 容器映像允許在 AMI 組建建立時進行昂貴的 I/O 操作 （檔案擷取），而不是容器啟動。因此，AMI 上將

擷取所有必要的影像層，並準備好可供使用，以加快 Windows 容器啟動的時間，並可以開始接受流量。在推送操作期間，只會將構成映像的圖層上傳到儲存庫。

下列範例顯示 Amazon ECR 上的 fluentd-windows-sac2004 映像只有 390.18MB。這是推送操作期間發生的上傳量。

下列範例顯示推送到 Amazon ECR 儲存庫的流暢 Windows ltsc 映像。存放在 ECR 中的圖層大小為 533.05MB。



以下來自 的輸出`docker image ls`，磁碟上 fluentd v1.14-windows-ltsc2019-1 的大小為 6.96GB，但這並不表示它下載並擷取了該數量的資料。

實際上，在提取操作期間，只會下載和擷取壓縮的 533.05MB。

```
REPOSITORY                                                                  TAG
            IMAGE ID       CREATED         SIZE
111122223333.dkr.ecr.us-east-1.amazonaws.com/fluentd-windows-coreltsc   latest
            721afca2c725   7 weeks ago     6.96GB
fluent/fluentd                                                              v1.14-windows-
ltsc2019-1   721afca2c725   7 weeks ago     6.96GB
amazonaws.com/eks/pause-windows                                             latest
            6392f69ae6e7   10 months ago   255MB
```

大小欄顯示影像的整體大小，6.96GB。將其分解：

- Windows Server Core 2019 LTSC 基礎映像 = 5.74GB

- Fluentd 未壓縮的基礎映像 = 6.96GB
- 磁碟差異 = 1.2GB
- Fluentd 壓縮最終映像 ECR = 533.05MB

本機磁碟上已存在基礎映像，導致磁碟上的總量為額外 1.2GB。下次在大小欄中看到 GBs 的數量時，請不要擔心太多，可能超過 70% 的 GB 已經在磁碟上做為快取的容器映像。

參考資料

使用 EC2 映像建置器和映像快取策略加速 Windows 容器啟動時間

執行異質工作負載

Kubernetes 支援異質叢集，您可以在相同叢集中混合使用 Linux 和 Windows 節點。在該叢集中，您可以混合在 Linux 上執行的 Pod 和在 Windows 上執行的 Pod。您甚至可以在同一個叢集中執行多個版本的 Windows。不過，做出此決策時，需要考慮幾個因素（如下所述）。

將 PODs 指派給節點最佳實務

為了將 Linux 和 Windows 工作負載保留在其各自的作業系統特定節點上，您需要使用節點選取器和污點/容錯的一些組合。在異質環境中排程工作負載的主要目標是避免中斷現有 Linux 工作負載的相容性。

確保作業系統特定的工作負載登陸適當的容器主機

使用者可以確保使用 nodeSelectors 在適當的主機上排程 Windows 容器。目前所有 Kubernetes 節點都有下列預設標籤：

```
kubernetes.io/os = [windows|linux]
kubernetes.io/arch = [amd64|arm64|...]
```

如果 Pod 規格不包含如 的 nodeSelector`"kubernetes.io/os": windows`，則可能會在任何主機、Windows 或 Linux 上排程 Pod。這可能有問題，因為 Windows 容器只能在 Windows 上執行，Linux 容器只能在 Linux 上執行。

在企業環境中，針對 Linux 容器擁有大量預先存在的部署，以及 Helm Chart 等off-the-shelf組態的生態系統並不罕見。在這些情況下，您可能會對部署的 nodeSelectors 進行變更感到遲疑。另一種方法是使用 Taints。

例如：`--register-with-taints='os=windows:NoSchedule'`

如果您使用的是 EKS，eksctl 提供透過 clusterConfig 套用污點的方法：

```
NodeGroups:
  - name: windows-ng
    amiFamily: WindowsServer2022FullContainer
    ...
    labels:
      nodeclass: windows2022
    taints:
      os: "windows:NoSchedule"
```

將污點新增至所有 Windows 節點，排程器不會在這些節點上排程 Pod，除非它們能容忍污點。Pod 資訊清單範例：

```
nodeSelector:
    kubernetes.io/os: windows
tolerations:
    - key: "os"
      operator: "Equal"
      value: "windows"
      effect: "NoSchedule"
```

處理相同叢集中的多個 Windows 組建

每個 Pod 使用的 Windows 容器基礎映像必須符合與節點相同的核心建置版本。如果您想要在相同叢集中使用多個 Windows Server 組建，則應設定其他節點標籤、nodeSelectors 或利用稱為 windows-build 的標籤。

Kubernetes 1.17 會自動新增標籤 node.kubernetes.io/windows-build，以簡化相同叢集中多個 Windows 組建的管理。如果您執行的是較舊的版本，建議您手動將此標籤新增至 Windows 節點。

此標籤反映需要符合相容性的 Windows 主要、次要和建置編號。以下是目前用於每個 Windows Server 版本的值。

請務必注意，Windows Server 正在移至長期服務頻道 (LTSC) 做為主要發行頻道。Windows Server 半年頻道 (SAC) 已於 2022 年 8 月 9 日淘汰。未來不會有 Windows Server 的 SAC 版本。

產品名稱	組建編號 (s)
伺服器完整 2022 LTSC	20348 年 10 月 0 日
伺服器核心 2019 LTSC	10.0.17763

您可以透過下列命令檢查作業系統建置版本：

```
kubectl get nodes -o wide
```

KERNEL-VERSION 輸出符合 Windows 作業系統建置版本。

```
NAME                           STATUS   ROLES    AGE   VERSION                INTERNAL-
IP   EXTERNAL-IP     OS-IMAGE                          KERNEL-VERSION
 CONTAINER-RUNTIME
ip-10-10-2-235.ec2.internal   Ready    <none>   23m   v1.24.7-eks-fb459a0
 10.10.2.235   3.236.30.157    Windows Server 2022 Datacenter   10.0.20348.1607
         containerd://1.6.6
ip-10-10-31-27.ec2.internal   Ready    <none>   23m   v1.24.7-eks-fb459a0
 10.10.31.27   44.204.218.24   Windows Server 2019 Datacenter   10.0.17763.4131
         containerd://1.6.6
ip-10-10-7-54.ec2.internal    Ready    <none>   31m   v1.24.11-eks-a59e1f0   10.10.7.54
    3.227.8.172     Amazon Linux 2                  5.10.173-154.642.amzn2.x86_64
 containerd://1.6.19
```

以下範例會將額外的 nodeSelector 套用至 Pod 資訊清單，以便在執行不同的 Windows 節點群組作業系統版本時符合正確的 Windows 建置版本。

```
nodeSelector:
    kubernetes.io/os: windows
    node.kubernetes.io/windows-build: '10.0.20348'
tolerations:
    - key: "os"
    operator: "Equal"
    value: "windows"
    effect: "NoSchedule"
```

使用 RuntimeClass 簡化 Pod 資訊清單中的 NodeSelector 和公差

您也可以使用 RuntimeClass 來簡化使用污點和容錯的程序。這可以透過建立用於封裝這些污點和容錯的 RuntimeClass 物件來完成。

透過執行下列資訊清單來建立 RuntimeClass：

```
apiVersion: node.k8s.io/v1beta1
kind: RuntimeClass
metadata:
  name: windows-2022
handler: 'docker'
scheduling:
  nodeSelector:
    kubernetes.io/os: 'windows'
    kubernetes.io/arch: 'amd64'
    node.kubernetes.io/windows-build: '10.0.20348'
  tolerations:
  - effect: NoSchedule
    key: os
    operator: Equal
    value: "windows"
```

建立 Runtimeclass 後，請使用 做為 Pod 資訊清單上的規格來指派：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iis-2022
  labels:
    app: iis-2022
spec:
  replicas: 1
  template:
    metadata:
      name: iis-2022
      labels:
        app: iis-2022
    spec:
      runtimeClassName: windows-2022
      containers:
      - name: iis
```

受管節點群組支援

為了協助客戶以更簡化的方式執行 Windows 應用程式，AWS 於 2022 年 12 月 15 日啟動對 Windows 容器的 Amazon EKS 受管節點群組 (MNG) 支援。為了協助協調營運團隊，Windows MNGs會使用與 Linux MNGs 相同的工作流程和工具啟用。支援 Windows Server 2019 和 2022 的完整和核心 AMI (Amazon Machine Image) 系列版本。

受管節點群組 (MNG) 支援下列 AMI 系列。

AMI 系列
WINDOWS_CORE_2019_x86_64
WINDOWS_FULL_2019_x86_64
WINDOWS_CORE_2022_x86_64
WINDOWS_FULL_2022_x86_64

其他文件

AWS 官方文件：https：//https://docs.aws.amazon.com/eks/latest/userguide/windows-support.html

若要進一步了解 Pod Networking (CNI) 的運作方式，請查看以下連結：https：//https://docs.aws.amazon.com/eks/latest/userguide/pod-networking.html

部署適用於 Windows on EKS 的受管節點群組的 AWS 部落格：https：//https://aws.amazon.com/blogs/containers/deploying-amazon-eks-windows-managed-node-groups/

Pod 安全內容

Pod 安全政策 (PSP) 和 Pod 安全標準 (PSS) 是在 Kubernetes 中強制執行安全性的兩種主要方式。請注意，自 PodSecurityPolicy 已棄用，並將在 v1.25 中移除，且 Pod 安全標準 (PSS) 是 Kubernetes 建議的方法，可強制執行安全性。

Pod 安全政策 (PSP) 是 Kubernetes 中實作安全政策的原生解決方案。PSP 是叢集層級的資源，可控制 Pod 規格的安全敏感層面。使用 Pod 安全政策，您可以定義一組 Pod 必須滿足的條件，才能由叢集接受。PSP 功能已從 Kubernetes 的早期開始提供，旨在封鎖在指定叢集上建立設定錯誤的 Pod。

如需 Pod 安全政策的詳細資訊，請參閱 Kubernetes 文件。根據 Kubernetes 棄用政策，較舊的版本將在功能棄用九個月後停止取得支援。

另一方面，Pod 安全標準 (PSS) 是建議的安全性方法，通常使用安全內容實作，在 Pod 資訊清單中定義為 Pod 和容器規格的一部分。PSS 是 Kubernetes 專案團隊為解決 Pod 的安全相關最佳實務而定義的官方標準。它定義了政策，例如基準（最低限制性、預設性）、特權（無限制性）和限制性（最高限制性）。

我們建議您從基準設定檔開始。PSS 基準設定檔在安全性和潛在摩擦之間提供穩固的平衡，需要最少的例外狀況清單，可作為工作負載安全性的良好起點。如果您目前正在使用 PSPs建議您切換到 PSS。如需 PSS 政策的詳細資訊，請參閱 Kubernetes 文件。這些政策可以使用多種工具強制執行，包括來自 OPA 和 Kyverno 的工具。例如，Kyverno 在此處提供 PSS 政策的完整集合。

安全內容設定可讓 提供選取程序的權限、使用程式描述檔限制個別程式的功能、允許權限提升、篩選系統呼叫等。

在安全性內容方面，Kubernetes 中的 Windows Pod 與標準 Linux 工作負載有一些限制和差異。

Windows 使用每個容器的任務物件搭配系統命名空間篩選條件，以包含容器中的所有程序，並提供與主機的邏輯隔離。如果沒有命名空間篩選，就無法執行 Windows 容器。這表示無法在主機的內容中宣告系統權限，因此 Windows 上無法使用特權容器。

以下是唯一記錄`windowsOptions`的 Windows 安全內容選項，其餘則是一般安全內容選項

如需 Windows 和 linux 支援的安全內容屬性清單，請參閱此處的官方文件。

Pod 特定設定會套用至所有容器。如果未指定，則會使用 PodSecurityContext 中的選項。如果同時在 SecurityContext 和 PodSecurityContext 中設定，則 SecurityContext 中指定的值優先。

例如，Pod 和容器的 runAsUserName 設定是 Windows 選項的大約相當於 Linux 特定 runAsUser 設定，並且在下列資訊清單中，Pod 特定安全內容會套用至所有容器

```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-pod-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
```

```
  nodeSelector:
    kubernetes.io/os: windows
```

但在下列情況下，容器層級安全內容會覆寫 Pod 層級安全內容。

```
apiVersion: v1
kind: Pod
metadata:
  name: run-as-username-container-demo
spec:
  securityContext:
    windowsOptions:
      runAsUserName: "ContainerUser"
  containers:
  - name: run-as-username-demo
    ..
    securityContext:
        windowsOptions:
            runAsUserName: "ContainerAdministrator"
  nodeSelector:
    kubernetes.io/os: windows
```

runAsUserName 欄位可接受的值範例：ContainerAdministrator、ContainerUser、NT AUTHORITY\NETWORK SERVICE、NT AUTHORITY\LOCAL SERVICE

使用 ContainerUser for Windows Pod 執行容器通常是個好主意。容器和主機之間不會共用使用者，但 ContainerAdministrator 在容器中具有 的其他權限。請注意，有使用者名稱限制需要注意。

使用 ContainerAdministrator 的良好範例是設定 PATH。您可以使用 USER 指令來執行此操作，如下所示：

```
USER ContainerAdministrator
RUN setx /M PATH "%PATH%;C:/your/path"
USER ContainerUser
```

另請注意，秘密會在節點磁碟區上以純文字撰寫 （相較於 linux 上的 tmpfs/記憶體內）。這表示您必須執行兩件事

- 使用檔案 ACLs 保護秘密檔案位置
- 使用 BitLocker 使用磁碟區層級加密

持久性儲存選項

什麼是樹狀內與out-of-tree磁碟區外掛程式？

在推出容器儲存界面 (CSI) 之前，所有磁碟區外掛程式都是在樹狀結構中，表示它們是建置、連結、編譯和隨附於核心 Kubernetes 二進位檔，並擴展核心 Kubernetes API。這表示將新的儲存系統新增至 Kubernetes （磁碟區外掛程式） 需要檢查核心 Kubernetes 程式碼儲存庫中的程式碼。

Out-of-tree磁碟區外掛程式是獨立於 Kubernetes 程式碼庫開發，並部署 （安裝） 在 Kubernetes 叢集上做為延伸模組。這可讓廠商更新out-of-band驅動程式，即與 Kubernetes 發行週期分開。這在很大程度上是可能的，因為 Kubernetes 已建立儲存介面或 CSI，為廠商提供與 k8 連接的標準方式。

您可以在 https://docs.aws.amazon.com/eks/latest/userguide/storage.html：// 查看有關 Amazon Elastic Kubernetes Services (EKS) 儲存類別和 CSI 驅動程式的詳細資訊

適用於 Windows 的樹狀內磁碟區外掛程式

Kubernetes 磁碟區可讓具有資料持久性需求的應用程式部署在 Kubernetes 上。持久性磁碟區的管理包含佈建/取消佈建/調整磁碟區大小、將磁碟區連接至 Kubernetes 節點或從 Kubernetes 節點移除磁碟區，以及將磁碟區掛載/卸載至 Pod 中的個別容器。針對特定儲存後端或通訊協定實作這些磁碟區管理動作的程式碼，會以 Kubernetes 磁碟區外掛程式 （樹狀內磁碟區外掛程式） 的形式運送。在 Amazon Elastic Kubernetes Services (EKS) 上，Windows 支援以下類別的 Kubernetes 磁碟區外掛程式：

樹狀內磁碟區外掛程式：awsElasticBlockStore

若要在 Windows 節點上使用樹狀內磁碟區外掛程式，必須建立額外的 StorageClass，才能使用 NTFS 做為 fsType。在 EKS 上，預設 StorageClass 使用 ext4 作為預設 fsType。

StorageClass 可讓管理員描述其提供的「類別」儲存體。不同的類別可能會對應至由叢集管理員決定 quality-of-service層級、備份政策或任意政策。Kubernetes 對於代表的類別不建議。此概念有時在其他儲存系統中稱為「設定檔」。

您可以執行下列命令來檢查：

```
kubectl describe storageclass gp2
```

輸出：

```
Name:            gp2
IsDefaultClass:  Yes
Annotations:     kubectl.kubernetes.io/last-applied-
configuration={"apiVersion":"storage.k8s.io/v1","kind":"StorageClas
","metadata":{"annotations":{"storageclass.kubernetes.io/is-default-
class":"true"},"name":"gp2"},"parameters":{"fsType"
"ext4","type":"gp2"},"provisioner":"kubernetes.io/aws-
ebs","volumeBindingMode":"WaitForFirstConsumer"}
,storageclass.kubernetes.io/is-default-class=true
Provisioner:           kubernetes.io/aws-ebs
Parameters:            fsType=ext4,type=gp2
AllowVolumeExpansion:  <unset>
MountOptions:          <none>
ReclaimPolicy:         Delete
VolumeBindingMode:     WaitForFirstConsumer
Events:                <none>
```

若要建立新的 StorageClass 以支援 NTFS，請使用下列資訊清單：

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gp2-windows
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  fsType: ntfs
volumeBindingMode: WaitForFirstConsumer
```

執行下列命令來建立 StorageClass：

```
kubectl apply -f NTFSStorageClass.yaml
```

下一個步驟是建立持久性磁碟區宣告 (PVC)。

PersistentVolume (PV) 是由管理員佈建或使用 PVC 動態佈建的叢集儲存體。它是叢集中的資源，就像節點是叢集資源一樣。此 API 物件會擷取儲存體實作的詳細資訊，包括 NFS、iSCSI 或cloud-provider-specific儲存系統。

PersistentVolumeClaim (PVC) 是使用者提出的儲存請求。宣告可以請求特定大小和存取模式（例如，可以掛載 ReadWriteOnce、ReadOnlyMany 或 ReadWriteMany)。

對於不同的使用案例，使用者需要具有不同屬性的 PersistentVolumes，例如效能。叢集管理員需要能夠提供各種相異於大小和存取模式的PersistentVolumes，而無需讓使用者了解這些磁碟區如何實作的詳細資訊。對於這些需求，有 StorageClass 資源。

在下面的範例中，已在命名空間視窗中建立 PVC。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-windows-pv-claim
  namespace: windows
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: gp2-windows
  resources:
    requests:
      storage: 1Gi
```

執行下列命令來建立 PVC：

```
kubectl apply -f persistent-volume-claim.yaml
```

下列資訊清單會建立 Windows Pod、將 VolumeMount 設定為 ，`C:\Data`並使用 PVC 做為 上連接的儲存體`C:\Data`。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: windows-server-ltsc2019
  namespace: windows
spec:
  selector:
    matchLabels:
      app: windows-server-ltsc2019
      tier: backend
      track: stable
  replicas: 1
  template:
    metadata:
      labels:
        app: windows-server-ltsc2019
```

```
        tier: backend
        track: stable
    spec:
      containers:
      - name: windows-server-ltsc2019
        image: mcr.microsoft.com/windows/servercore:ltsc2019
        ports:
        - name: http
          containerPort: 80
        imagePullPolicy: IfNotPresent
        volumeMounts:
        - mountPath: "C:\\data"
          name: test-volume
      volumes:
        - name: test-volume
          persistentVolumeClaim:
            claimName: ebs-windows-pv-claim
      nodeSelector:
        kubernetes.io/os: windows
        node.kubernetes.io/windows-build: '10.0.17763'
```

透過 PowerShell 存取 Windows Pod 來測試結果：

```
kubectl exec -it podname powershell -n windows
```

在 Windows Pod 中，執行： `ls`

輸出：

```
PS C:\> ls


    Directory: C:\


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          3/8/2021   1:54 PM                data
d-----          3/8/2021   3:37 PM                inetpub
d-r---          1/9/2021   7:26 AM                Program Files
d-----          1/9/2021   7:18 AM                Program Files (x86)
d-r---          1/9/2021   7:28 AM                Users
d-----          3/8/2021   3:36 PM                var
```

```
d-----          3/8/2021   3:36 PM                Windows
-a----         12/7/2019   4:20 AM           5510 License.txt
```

資料目錄由 EBS 磁碟區提供。

Out-of-tree

與 CSI 外掛程式相關聯的程式碼會以out-of-tree指令碼和二進位形式提供，通常以容器映像形式分佈，並使用 DaemonSets 和 StatefulSets 等標準 Kubernetes 建構模組進行部署。CSI 外掛程式可在 Kubernetes 中處理各種磁碟區管理動作。CSI 外掛程式通常由節點外掛程式（以 DaemonSet 的形式在每個節點上執行）和控制器外掛程式組成。

CSI 節點外掛程式（特別是與做為區塊型儲存設備或透過共用檔案系統公開的持久性磁碟區相關聯的外掛程式）需要執行各種特殊權限操作，例如掃描磁碟裝置、掛載檔案系統等。這些操作會因每個主機作業系統而有所不同。對於 Linux 工作者節點，容器化 CSI 節點外掛程式通常會部署為特權容器。對於 Windows 工作者節點，使用 csi-proxy 支援容器化 CSI 節點外掛程式的特權操作，csi-proxy 是需要在每個 Windows 節點上預先安裝的社群受管、獨立二進位檔。

Amazon EKS Optimized Windows AMI 從 2022 年 4 月開始包含 CSI-proxy。客戶可以使用 Windows 節點上的 SMB CSI 驅動程式來存取 Amazon FSx for Windows File Server、Amazon FSx for NetApp ONTAP SMB Shares 和/或 AWS Storage Gateway — File Gateway。

下列部落格提供實作詳細資訊，說明如何設定 SMB CSI 驅動程式使用 Amazon FSx for Windows File Server 做為 Windows Pod 的持久性儲存體。

Amazon FSx for Windows File Server

選項是透過稱為 SMB Global Mapping 的 SMB 功能使用 Amazon FSx for Windows File Server，這可讓您在主機上掛載 SMB 共用，然後將共用上的目錄傳遞到容器中。容器不需要設定特定的伺服器、共用、使用者名稱或密碼 - 它們都是在主機上處理。容器的運作方式與具有本機儲存體相同。

> SMB 全域映射對協調器來說是透明的，並且透過 HostPath 掛載，這可能隱含在安全考量中。

在下面的範例中，路徑`G:\Directory\app-state`是 Windows 節點上的 SMB 共享。

```
apiVersion: v1
kind: Pod
metadata:
  name: test-fsx
```

```
spec:
  containers:
  - name: test-fsx
    image: mcr.microsoft.com/windows/servercore:ltsc2019
    command:
      - powershell.exe
      - -command
      - "Add-WindowsFeature Web-Server; Invoke-WebRequest -UseBasicParsing
 -Uri 'https://dotnetbinaries.blob.core.windows.net/servicemonitor/2.0.1.6/
ServiceMonitor.exe' -OutFile 'C:\\ServiceMonitor.exe'; echo '<html><body><br/
><br/><marquee><H1>Hello EKS!!!<H1><marquee></body><html>' > C:\\inetpub\\wwwroot\
\default.html; C:\\ServiceMonitor.exe 'w3svc'; "
    volumeMounts:
      - mountPath: C:\dotnetapp\app-state
        name: test-mount
  volumes:
    - name: test-mount
      hostPath:
        path: G:\Directory\app-state
        type: Directory
  nodeSelector:
      beta.kubernetes.io/os: windows
      beta.kubernetes.io/arch: amd64
```

下列部落格提供如何將 Amazon FSx for Windows File Server 設定為 Windows Pod 持久性儲存體的實作詳細資訊。

強化 Windows 容器映像

您是否正在強化 Windows 容器映像？ 多年來，我與全球客戶合作，協助他們將舊版工作負載遷移至容器，特別是 Windows 工作負載。憑藉 20 多年的經驗，我看到組織投入大量心力和資源來強化 Windows Server，實作從 CIS Benchmarks 到執行期防毒保護的所有項目，以保護敏感資料。

不過，出現了令人擔憂的趨勢。由於這些高度安全的虛擬機器會現代化為容器，因此許多重要的強化實務都會遭到忽略。從基礎映像 (OS) 到 IIS 等 Web 服務，Windows 安全最佳實務通常被忽略，其中大部分重點僅放在保護容器主機。請務必了解，當容器在隔離的命名空間中運作時，它們仍然會與主機共用核心基本概念。攻擊者通常對橫向移動更感興趣，而不是直接鎖定容器主機，使其能夠利用脆弱的容器安全設定並存取敏感資料。

文件的目標是強調您應該特別針對 IIS 上託管ASP.NET網站的 Windows 容器實作的一些基本安全設定。我們將專注於四個關鍵領域：

- 帳戶安全政策
- 稽核政策
- IIS 安全最佳實務
- 最低權限準則

首先，我們將深入探討為什麼每個安全組態對於保護您的 Windows 容器至關重要，並檢查它們減輕的特定風險及其提供的安全優勢。接下來，我們將逐步解說程式碼片段，示範如何在 Dockerfile 中正確實作這些組態，確保您的容器已強化，防範潛在威脅。最後，我們將詳細說明每個設定，提供其功能的完整說明、對容器安全性的影響，以及它如何有助於保護您的應用程式。此方法不僅會示範如何套用這些最佳實務，也會讓您深入了解為何這些做法對於在容器化環境中維持強大的安全狀態至關重要。

1. 使用本機安全政策和登錄檔設定帳戶政策（密碼或鎖定）

Windows Server Core 是最小的安裝選項，可作為 【EKS Optimized Windows AMI】(https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-windows-ami.html：//) 的一部分使用。使用本機安全政策及登錄檔設定帳戶政策（密碼或鎖定）可強制執行強大的密碼和鎖定規則，以強化系統安全。這些政策要求使用者建立具有已定義最小長度和複雜性的高強度密碼，以防止常見的密碼相關攻擊。

透過設定密碼存留期上限，系統會提示使用者定期更新密碼，從而降低登入資料洩露的可能性。鎖定政策透過在指定的失敗登入嘗試次數後暫時鎖定帳戶來新增額外的保護層，有助於防止暴力攻擊。透過 Windows 登錄檔設定這些設定，可讓管理員在系統層級強制執行這些安全措施，確保整個組織的一致性和合規性。在 Windows 容器中套用這些帳戶政策對於維持安全性一致性至關重要，即使容器通常是暫時性的，且適用於隔離的工作負載：

安全一致性

- 合規：在容器中強制執行一致的密碼政策和鎖定規則有助於維持安全合規，尤其是在需要嚴格存取控制的環境中（例如 HIPAA、PCI-DSS 等法規合規）。
- 強化容器：套用這些設定可確保您的 Windows 容器強化，防範未經授權的存取或密碼型攻擊，使容器的安全狀態與更廣泛的系統安全政策保持一致。

防禦暴力攻擊

- 帳戶鎖定：這些設定可在登入嘗試失敗的特定次數之後鎖定帳戶，協助防止暴力登入嘗試。這可防止攻擊者嘗試不限數量的密碼。

- 密碼複雜性：需要長度足夠的複雜密碼，可以降低使用弱密碼的可能性，即使在隔離的容器化環境中也是如此。

多使用者案例

- 如果您的容器化應用程式旨在處理多個使用者或需要使用者身分驗證，強制執行密碼政策可確保容器內的使用者帳戶遵守嚴格的安全規則，限制只有授權使用者才能存取。

持久性 Windows 容器

- 雖然容器通常被視為暫時性，但某些 Windows 容器可以執行長期服務或處理使用者管理，因此請務必強制執行與一般 Windows 伺服器類似的適當安全政策。

混合環境中的一致性

- 如果您在基礎設施中同時執行虛擬機器和容器，在所有環境中套用相同的安全政策（例如密碼/鎖定政策）可確保一致的安全標準，簡化控管和管理。

總而言之，在 Windows 容器內套用這些帳戶政策可確保您的容器不是安全策略的弱點，可防範密碼攻擊，並在整個環境中強制執行一致性。

Dockerfile：

```
# Configure account policies for password complexity and lockout
RUN powershell -Command \
      "Write-Output 'Configuring Account Policies (Password/Lockout)...'; \
      NET ACCOUNTS /MINPWLEN:14 /MAXPWAGE:60 /MINPWAGE:14 /LOCKOUTTHRESHOLD:5
```

說明：

本節透過 Windows 登錄檔設定密碼和鎖定設定的帳戶政策。這些政策透過控制密碼要求和帳戶鎖定閾值來協助強制執行安全性。

1. MinimumPasswordLength (MINPWLEN) = 14 此設定會定義密碼的字元數下限。範圍為 0-14 個字元；預設值為六個字元。
2. MaximumPasswordAge (MAXPWAGE) = 60 此設定會定義密碼有效的天數上限。使用 UNLIMITED 不會指定任何限制。/MAXPWAGE 不能小於 /MINPWAGE。範圍為 1-999；預設值為 90 天

3. 鎖定閾值 (LOCKOUTTHRESHOLD) = 5 此設定會定義失敗登入嘗試的閾值。5 次不正確的嘗試後，帳戶將被鎖定。

這些設定可透過強制執行強式密碼政策，並在登入嘗試失敗一定次數後鎖定帳戶，協助改善密碼安全性並防止暴力攻擊。

2. 稽核政策

稽核政策對於 Windows Containers 很重要，因為它們提供安全事件的關鍵可見性，例如登入嘗試和權限使用、協助偵測未經授權的存取、監控使用者活動，並確保符合法規標準。即使在容器的暫時性本質下，稽核日誌對於事件調查、主動威脅偵測，以及跨容器化環境維持一致的安全狀態至關重要。

安全性監控與合規：

- 追蹤使用者活動：稽核政策可讓管理員監控容器內的使用者活動，例如登入嘗試和權限使用。這對於偵測未經授權的存取或可疑行為至關重要。
- 法規合規：許多組織需要記錄安全事件以符合 HIPAA、PCI-DSS 和 GDPR 等法規。在容器中啟用稽核政策可確保您符合這些要求，即使在容器化環境中也是如此。

事件調查：

- 鑑識和分析：如果容器化應用程式或服務遭到入侵，稽核日誌可以為事件後分析提供有價值的洞見。它們可協助安全團隊追蹤攻擊者採取的動作，或識別違規的發生方式。
- 即時偵測：稽核日誌可讓管理員設定關鍵事件的即時提醒（例如，失敗的登入嘗試、權限提升）。此主動監控有助於及早偵測攻擊，並加快回應時間。

跨環境的一致性：

- 統一安全狀態：透過登錄在容器中套用稽核政策，您可以確保容器化和非容器化環境的安全實務一致。這可避免容器成為安全監控的盲點。
- 混合環境中的可見性：對於同時執行傳統 Windows 伺服器和容器的組織，稽核政策在所有平台上提供類似的可見性和控制，讓管理更容易且更有效率。

追蹤特殊權限操作：

- 權限使用稽核：在以更高權限執行應用程式或執行管理任務的容器環境中，稽核權限操作可確保責任。您可以記錄存取敏感資源或在容器內執行關鍵任務的人員。

- 防止濫用權限：透過監控權限使用，您可以偵測未經授權的使用者何時嘗試提升其權限或存取容器中的限制區域，這有助於防止內部或外部攻擊。

偵測未經授權的存取嘗試：

- 失敗的登入嘗試：為失敗的登入嘗試啟用稽核政策有助於識別暴力攻擊或未經授權嘗試存取容器化應用程式。這可讓您了解誰嘗試存取系統，以及存取系統的頻率。
- 帳戶鎖定監控：稽核帳戶鎖定事件可讓管理員偵測和調查由可疑或惡意活動引起的潛在鎖定。

即使在偶發環境中，持久性安全：

- 暫時性但安全：雖然容器是暫時性的，這表示可以經常刪除並重新建立，但稽核仍然扮演關鍵角色，以確保在容器執行時擷取安全事件。這可確保在容器生命週期期間記錄關鍵安全事件。

集中式記錄：

- 將日誌轉送至集中式系統：容器可與集中式記錄系統（例如 ELK 堆疊、AWS CloudWatch) 整合，以從多個容器執行個體擷取稽核日誌。這可讓您在基礎設施間更好地分析和相互關聯安全事件。

Dockerfile：

```
# Configure audit policies for logging security events
RUN powershell -Command \
    "Write-Host 'Configuring Audit Policy..'; \
    Set-ItemProperty -Path 'HKLM:\\SYSTEM\\CurrentControlSet\\Control\\Lsa' -Name
 'SCENoApplyLegacyAuditPolicy' -Value 0; \
    auditpol /set /category:"Logon/Logoff" /subcategory:"Logon" /failure:enable

# Creates STDOUT on Windows Containers (check GitHub LogMonitor:: https://github.com/
microsoft/windows-container-tools/blob/main/LogMonitor/README.md)
COPY LogMonitor.exe LogMonitorConfig.json 'C:\\LogMonitor\\'
WORKDIR /LogMonitor
```

說明：

本節使用登錄檔修改來設定稽核政策。稽核政策會控制 Windows 記錄的安全事件，這有助於監控和偵測未經授權的存取嘗試。

1. SCENoApplyLegacyAuditPolicy = 0 這會停用舊版稽核政策格式，讓較新版本的 Windows 引入更精細的稽核政策。這對現代稽核組態很重要。
2. Auditpol 子類別：「登入」 此設定可同時稽核成功和失敗登入事件。值 3 表示 Windows 將記錄成功和失敗的登入嘗試。這有助於監控誰正在存取系統並擷取失敗的登入嘗試。

這些稽核政策對於安全監控和合規至關重要，因為它們提供重要安全事件的詳細日誌，例如登入嘗試和使用特殊權限操作。

3. Windows 容器的 IIS 安全最佳實務

在 Windows Containers 中實作 IIS 最佳實務非常重要，原因有幾個，可確保您的應用程式安全、高效能且可擴展。雖然容器提供隔離和輕量型環境，但它們仍需要適當的組態，以避免漏洞和操作問題。以下是為什麼在 Windows Containers 中遵循 IIS 的最佳實務至關重要：

安全性

- 防止常見漏洞：IIS 通常是跨網站指令碼 (XSS)、點擊劫持和資訊揭露等攻擊的目標。實作安全標頭（例如 X-Content-Type-Options、X-Frame-Options 和 Strict-Transport-Security) 有助於保護您的應用程式免受這些威脅。
- 隔離不足：容器已隔離，但設定錯誤的 IIS 執行個體可能會公開敏感資訊，例如伺服器版本詳細資訊、目錄清單或未加密的通訊。透過停用目錄瀏覽和移除 IIS 版本標頭等功能，您可以將攻擊面降至最低。
- 加密和 HTTPS：最佳實務，例如強制執行僅限 HTTPS 的連線，確保傳輸中的資料已加密，防止敏感資訊遭到攔截。

效能

- 高效率資源使用：IIS 最佳實務，例如啟用動態和靜態壓縮，可減少頻寬使用量並縮短載入時間。這些最佳化在容器化環境中特別重要，其中資源會跨容器和主機系統共用。
- 最佳化記錄：正確設定記錄（例如，包括 X-Forwarded-For 標頭）可確保您可以追蹤用戶端活動，同時將不必要的記錄開銷降至最低。這可協助您收集相關資料進行故障診斷，而不會降低效能。

可擴展性和可維護性

- 跨環境一致性：透過遵循最佳實務，您可以確保 IIS 組態在多個容器執行個體之間保持一致。這可簡化擴展，並確保在部署新容器時，它們遵循相同的安全性和效能準則。

- 自動化組態：Dockerfiles 中的最佳實務，例如設定資料夾許可和停用不必要的功能，確保每個新容器都已正確設定。這可降低手動介入並降低人為錯誤的風險。

合規

- 符合法規要求：許多產業都有嚴格的法規要求（例如 PCI-DSS、HIPAA)，要求特定的安全措施，例如加密通訊 (HTTPS) 和用戶端請求的記錄。遵循容器中的 IIS 最佳實務有助於確保符合這些標準。
- 可稽核性：實作稽核政策和安全記錄允許事件的可追蹤性，這在稽核中至關重要。例如，記錄 X-Forwarded-For 標頭可確保用戶端 IP 地址正確記錄在以代理為基礎的架構中。

將共用環境中的風險降至最低

- 避免設定錯誤：容器共用主機的核心，當它們彼此隔離時，設定不良的 IIS 執行個體可能會公開漏洞或造成效能瓶頸。最佳實務可確保每個 IIS 執行個體以最佳方式執行，降低跨容器問題的風險。
- 最低權限存取：為容器內的資料夾和檔案設定適當的許可（例如，在 PowerShell 中使用 Set-Acl)，可確保容器內的使用者和程序只有必要的存取權，進而降低權限提升或資料竄改的風險。

暫時性環境中的彈性

- 容器的暫時性：容器通常短暫且經常重建。套用 IIS 最佳實務可確保安全且一致地設定每個容器，無論重新部署多少次。這可防止隨著時間的推移引入錯誤組態。
- 緩解潛在的設定錯誤：透過自動強制執行最佳實務（例如停用較弱的通訊協定或標頭），容器重新啟動或更新期間設定錯誤的風險就會降到最低。

Dockerfile：

```
# Enforce HTTPS (disable HTTP) -- Only if container is target for SSL termination
RUN powershell -Command \
    "$httpBinding = Get-WebBinding -Name 'Default Web Site' -Protocol http | Where-
Object { $_.bindingInformation -eq '*:80:' }; \
    if ($httpBinding) { Remove-WebBinding -Name 'Default Web Site' -Protocol http -Port
 80; } \
    $httpsBinding = Get-WebBinding -Name 'Default Web Site' -Protocol https | Where-
Object { $_.bindingInformation -eq '*:443:' }; \
    if (-not $httpsBinding) { New-WebBinding -Name 'Default Web Site' -Protocol https -
Port 443 -IPAddress '*'; }"

# Use secure headers
```

```
RUN powershell -Command \
    "Write-Host 'Adding security headers...'; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 'system.applicationHost/sites/siteDefaults/logFile/customFields' -name
 "." -value @{logFieldName='X-Forwarded-For';sourceName='X-Forwarded-
For';sourceType='RequestHeader'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='Strict-
Transport-Security';value='max-age=31536000; includeSubDomains'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-Content-Type-
Options';value='nosniff'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-XSS-
Protection';value='1; mode=block'}; \
    Add-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpProtocol/customHeaders" -name "." -value @{name='X-Frame-
Options';value='DENY'};"

# Disable IIS version disclosure
RUN powershell -Command \
    "Write-Host 'Disabling IIS version disclosure...'; \
    Import-Module WebAdministration; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/security/requestFiltering" -name "removeServerHeader" -value
 "true";"

# Set IIS Logging Best Practices
RUN powershell -Command \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/directoryBrowse" -name "enabled" -value "false"; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/httpErrors" -name "existingResponse" -value "PassThrough"; \

# Enable IIS dynamic and static compression to optimize performance
RUN powershell -Command \
    "Write-Host 'Enabling IIS compression...'; \
    Enable-WindowsOptionalFeature -Online -FeatureName IIS-HttpCompressionDynamic; \
    Import-Module WebAdministration; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/urlCompression" -name "doDynamicCompression" -value "true"; \
    Set-WebConfigurationProperty -pspath 'MACHINE/WEBROOT/APPHOST' -filter
 "system.webServer/urlCompression" -name "doStaticCompression" -value "true"
```

```
# Ensure proper folder permissions using PowerShell's Set-Acl

RUN powershell -Command \
    "Write-Host 'Setting folder permissions for IIS...'; \
    $path = 'C:\\inetpub\\wwwroot'; \
    $acl = Get-Acl $path; \
    $iusr = New-Object System.Security.Principal.NTAccount('IIS_IUSRS'); \
    $rule = New-Object System.Security.AccessControl.FileSystemAccessRule($iusr,
 'ReadAndExecute', 'ContainerInherit, ObjectInherit', 'None', 'Allow'); \
    $acl.SetAccessRule($rule); \
    $users = New-Object System.Security.Principal.NTAccount('Users'); \
    $rule2 = New-Object System.Security.AccessControl.FileSystemAccessRule($users,
 'ReadAndExecute', 'ContainerInherit, ObjectInherit', 'None', 'Allow'); \
    $acl.SetAccessRule($rule2); \
    Set-Acl -Path $path -AclObject $acl"
```

說明：

此命令會將 IIS 設定為記錄 X-Forwarded-For 標頭，該標頭通常用於在請求通過代理或負載平衡器時擷取原始用戶端 IP 地址。根據預設，IIS 只會記錄負載平衡器或反向代理的 IP 地址，因此新增此自訂日誌欄位有助於追蹤真正的用戶端 IP 以進行安全稽核、分析和故障診斷。

1. X-Forwarded-For 標頭，通常用於在請求通過代理或負載平衡器時擷取原始用戶端 IP 地址。根據預設，IIS 只會記錄負載平衡器或反向代理的 IP 地址，因此新增此自訂日誌欄位有助於追蹤真正的用戶端 IP 以進行安全稽核、分析和故障診斷。
2. Strict-Transport-Security (HSTS) 確保瀏覽器僅透過 HTTPS 通訊。max-age=31536000 指定此政策強制執行 1 年，且 includeSubDomains 會將政策套用至所有子網域。
3. X-Content-Type-Options 防止瀏覽器「MIME-sniffing」回應離開宣告的 Content-Type。這有助於防止某些類型的攻擊。
4. X-XSS-Protection 在瀏覽器中啟用跨網站指令碼 (XSS) 保護。
5. X-Frame-Options 防止頁面內嵌在 iframe 中，防止點擊劫持攻擊。
6. 停用 IIS 版本揭露 此命令會在 HTTP 回應中停用伺服器標頭，預設會顯示正在使用的 IIS 版本。隱藏此資訊有助於降低攻擊者識別和鎖定 IIS 版本特定漏洞的風險。
7. 啟用僅限 HTTPS 的連線 此（註解）區段會強制執行 HTTPS 連線並停用 HTTP。如果未註解，Dockerfile 會將 IIS 設定為僅在連接埠 443 (HTTPS) 上接聽，並移除連接埠 80 上的預設 HTTP 繫結。這在終止容器內的 SSL 時非常有用，並確保所有流量都已加密。
8. 停用目錄瀏覽 當沒有預設文件時，防止 IIS 顯示目錄清單。這可避免將內部檔案結構公開給使用者。

9. 傳遞自訂錯誤頁面 確保如果應用程式有自己的錯誤處理，IIS 會讓應用程式的錯誤頁面傳遞，而不是顯示預設的 IIS 錯誤頁面。
10. 詳細錯誤模式 設定 IIS 僅顯示本機請求的詳細錯誤訊息，協助開發人員診斷問題，而不會向外部使用者公開敏感資訊。
11. 確保適當的資料夾許可 此區塊會設定 IIS Web 根目錄 (C：\inetpub\wwwroot) 的資料夾許可。它會設定 IIS_IUSRS 和使用者群組的讀取和執行許可，確保這些使用者可以存取 資料夾，但不能修改檔案。設定正確的許可可將未經授權的存取或竄改 Web 伺服器託管的檔案的風險降至最低。

遵循 Windows Containers 中的 IIS 最佳實務可確保容器化應用程式安全、高效能且可擴展。這些實務有助於防止漏洞、最佳化資源用量、確保合規性，以及維持跨容器執行個體的一致性。即使容器設計為隔離，仍需要適當的組態，才能將風險降至最低，並確保應用程式在生產環境中的可靠性。

4. 最低權限準則

最低權限原則 (PoLP) 對於 Windows 容器至關重要，原因有幾個，尤其是在增強安全性和最大限度地降低容器化環境中的風險。此原則指示系統或應用程式應以正常運作所需的最低許可層級運作。以下是為什麼它在 Windows 容器中很重要：

將攻擊面降至最低

- 容器通常會執行與各種系統元件互動的應用程式，以及應用程式擁有的更多權限，對這些元件的存取越廣泛。透過將容器的許可限制為僅必要，PoLP 可大幅減少攻擊面，讓攻擊者在容器遭到入侵時更難利用容器。

限制遭入侵容器的影響

- 如果 Windows 容器遭到入侵，執行具有過多權限（例如，管理員或根層級存取）的應用程式可能會允許攻擊者控制關鍵系統檔案或提升容器主機的權限。透過強制執行 PoLP，即使容器遭到入侵，攻擊者也會限制他們可以執行的動作，以防止進一步升級和存取敏感資源或其他容器。

多租戶環境中的保護

- 在雲端或企業環境中，多個容器可以在相同的實體或虛擬基礎設施上執行。PoLP 可確保遭入侵的容器無法存取屬於其他租用戶的資源或資料。此隔離對於在共用的多租用戶環境中維護安全性至關重要，可防止容器之間的橫向移動。

緩解權限提升

- 攻擊者可以使用具有高權限的容器來提升系統中的權限。PoLP 透過限制容器對系統資源的存取來降低此風險，從而防止未經授權的動作或權限提升超出容器的環境。

合規與稽核

- 許多法規標準和安全架構（例如 PCI DSS、HIPAA、GDPR) 要求系統遵守 PoLP，以限制對敏感資料的存取。執行具有限制權限的 Windows 容器有助於組織遵守這些法規，並確保應用程式只能存取他們特別需要的資源。

降低設定錯誤的風險

- 當容器以不必要的權限執行時，即使是輕微的錯誤設定也可能導致嚴重的安全漏洞。例如，如果以管理員身分執行的容器意外暴露到網際網路，攻擊者可能會取得系統的控制權。PoLP 透過預設為有限權限，使錯誤組態較不危險，協助防止此類風險。

改善容器安全狀態

- 透過遵循 PoLP，容器與基礎主機系統和彼此隔離更佳。這可確保容器化應用程式不太可能在定義的範圍之外存取或修改系統檔案或程序，從而保持主機作業系統和其他工作負載的完整性。

Dockerfile：

```
# Strongly recommended that when deploying a Windows server container to any multi-
tenant environment that your application runs via the ContainerUser account
USER ContainerUser
```

說明：

在本節中，USER ContainerUser 命令指定 Windows 容器內的應用程式應該在 ContainerUser 帳戶下執行，而不是預設的管理員帳戶。

以下是為什麼這很重要，特別是在多租戶環境中：

1. 最低權限原則：ContainerUser 帳戶是具有有限權限的非管理使用者。在此帳戶下執行應用程式會遵守最低權限原則，這有助於將利用風險降至最低。如果攻擊者入侵應用程式，他們對系統的存取將有限，從而減少潛在的損害。

2. 增強安全性：在多租戶環境中，容器可以共用相同的基礎基礎設施。以 ContainerUser 執行可確保即使一個容器遭到入侵，它也不會具有存取或修改重要系統檔案或其他容器的管理權限。這可大幅減少攻擊面。
3. 避免根存取：根據預設，容器的執行許可可能會提高（類似於 Linux 容器中的根存取），如果遭到利用，可能會很危險。使用 ContainerUser 可確保應用程式不會以不必要的管理權限執行，讓攻擊者更難提升權限。
4. 多租戶環境的最佳實務：在多個使用者或組織共用相同基礎設施（例如雲端）的環境中，安全性至關重要。執行具有限制許可的應用程式可防止租用戶的應用程式影響其他人，保護整個平台的敏感資料和資源。

USER ContainerUser 命令可確保應用程式以最低權限執行，藉由限制容器遭到入侵時可能造成的損壞來增強多租戶環境中的安全性。這是防止容器化環境中未經授權的存取或權限提升的最佳實務。

最低權限原則對於 Windows 容器至關重要，因為它會限制安全漏洞的潛在影響、減少攻擊面，並防止未經授權存取關鍵系統元件。透過執行僅具有必要許可的容器化應用程式，組織可以大幅增強其容器環境的安全性和穩定性，尤其是在多租戶和共用基礎設施中。

最終想法：為什麼保護您的 Windows 容器是現今威脅環境的必要條件

在現今快速發展的數位世界中，威脅越來越複雜且豐富，保護您的 Windows 容器不僅是建議，也是絕對必要的。容器提供輕量、彈性的方式來封裝和部署應用程式，但無法防範安全漏洞。隨著越來越多的企業採用容器來簡化其基礎設施，如果未適當保護，它們也會成為網路攻擊的潛在目標。

網際網路充滿各種威脅，從針對未修補漏洞的惡意執行者，到自動機器人掃描組態錯誤。如果沒有適當的安全措施，可以利用容器來公開敏感資料、提升權限，或做為攻擊的進入點，這些攻擊可能會危害更廣泛的基礎設施。這使得容器的安全性與保護您環境的任何其他部分一樣重要。

使用 Windows 容器時，許多傳統的安全最佳實務仍然適用。實作強大的帳戶政策、保護 IIS 組態、強制執行 HTTPS、使用嚴格的防火牆規則，以及對關鍵檔案套用最低權限存取，都是確保容器對攻擊保持彈性的關鍵措施。此外，定期稽核和記錄可提供容器內所發生情況的可見性，讓您在可疑活動變成完全爆發的事件之前，先加以捕捉。

保護 Windows 容器也符合法規要求，要求保護敏感資料並確保應用程式的完整性。隨著雲端原生和容器化架構越來越普遍，從基礎映像到執行中的容器，確保每一層的安全性，將有助於保護您的操作和維護客戶信任。

總而言之，容器化應用程式的崛起，加上越來越多的網路威脅，使得容器安全成為現代基礎設施管理不可轉移的層面。透過遵守最佳實務並持續監控漏洞，企業可以享受 Windows 容器的靈活性和效率，

而不會犧牲安全性。在這個威脅豐富的環境中，保護您的 Windows 容器不只是一個選項，也是必要項目。

混合部署的最佳實務

本指南提供使用 EKS 混合節點或 EKS Anywhere 在內部部署或邊緣環境中執行部署的指引。

我們目前已發佈下列主題的指南：

- EKS 混合節點和網路連線中斷的最佳實務

EKS 混合節點和網路連線中斷

對於習慣完全在其資料中心或節點中執行本機 Kubernetes 叢集的客戶來說，EKS 混合節點架構可能是新的。使用 EKS 混合節點時，Kubernetes 控制平面會在 AWS 區域中執行，只有節點會在內部部署執行，導致「延伸」或「延伸」Kubernetes 叢集架構。

這會導致一個常見問題：「如果我的節點與 Kubernetes 控制平面中斷連線會發生什麼情況？」

在本指南中，我們透過檢閱下列主題來回答這個問題。建議您透過網路連線來驗證應用程式的穩定性和可靠性，因為每個應用程式的行為可能根據其相依性、組態和環境而有所不同。如需測試設定、程序和結果，請參閱 aws-samples/eks-hybrid-examples GitHub 儲存庫，以使用 EKS 混合節點和您自己的應用程式來測試網路連線中斷。GitHub 儲存庫也包含測試的其他詳細資訊，用於驗證本指南中說明的行為。

- 透過網路連線的穩定性最佳實務
- 透過網路連線的 Kubernetes Pod 容錯移轉行為
- 透過網路連線的應用程式網路流量
- 透過網路連線的主機登入資料

透過網路連線的穩定性最佳實務

高可用性的聯網

避免混合節點與 Kubernetes 控制平面之間中斷網路連線的最佳方法是使用備援、彈性的內部部署環境與 AWS 之間的連線。如需使用這些解決方案建構高可用性混合網路的詳細資訊，請參閱 AWS Direct Connect 彈性工具組和 AWS Site-to-Site VPN 文件。

高度可用的應用程式

架構應用程式時，請考慮您的故障網域和不同類型的中斷的影響。Kubernetes 提供內建機制，可跨節點、區域和區域網域部署和維護應用程式複本。這些機制的使用取決於您的應用程式架構、環境和可用性需求。例如，無狀態應用程式通常可以使用多個複本部署，並且可以在任意主機和基礎設施容量之間移動，而且您可以使用節點選擇器和拓撲分散限制條件，在不同網域之間執行應用程式的執行個體。如需在 Kubernetes 上建置彈性應用程式的應用程式層級技術詳細資訊，請參閱 EKS 最佳實務指南。

Kubernetes 會在決定是否將 Pod 移至其他節點時，評估與 Kubernetes 控制平面中斷連接的節點的區域資訊。如果區域中的所有節點都無法連線，Kubernetes 會取消該區域中節點的 Pod 移出。最佳實務是，如果您部署的節點在多個資料中心或實體位置執行，請根據其資料中心或實體位置將區域指派給每個節點。當您使用雲端中的節點執行 EKS 時，AWS cloud-controller-manager 會自動套用此區域標籤。不過，cloud-controller-manager 不會與混合節點搭配使用，因此您可以透過 kubelet 組態傳遞此資訊。如何在混合節點的節點組態中設定區域的範例如下所示。當您使用混合節點 CLI () 將混合節點連接到叢集時，會傳遞組態nodeadm。如需`topology.kubernetes.io/zone`標籤的詳細資訊，請參閱 Kubernetes 文件。如需混合節點 CLI 的詳細資訊，請參閱混合節點節點廣告參考。

```
apiVersion: node.eks.aws/v1alpha1
kind: NodeConfig
spec:
  cluster:
    name: my-cluster
    region: my-region
  kubelet:
    flags:
       - --node-labels=topology.kubernetes.io/zone=dc1
  hybrid:
    ...
```

網路監控

如果您使用 AWS Direct Connect 或 AWS Site-to-Site VPN 進行混合連線，您可以利用 CloudWatch 警示、日誌和指標來觀察混合連線的狀態並診斷問題。如需詳細資訊，請參閱監控 AWS Direct Connect 資源和監控 AWS Site-to-Site VPN 連接。

建議為在 EKS 控制平面上執行的node-lifecycle-controller所報告`NodeNotReady`的事件建立警示，以表示混合節點可能正在發生網路連線中斷。您可以為 Controller Manager 啟用 EKS 控制平面記錄，並在 CloudWatch 中為狀態為「NodeNotReady」的「節點記錄狀態變更事件訊息」訊息建立指標篩選條件，以建立此警示。建立指標篩選條件後，您可以根據所需的閾值為此篩選條件建立警示。如需詳細資訊，請參閱 CloudWatch 文件中的警示日誌。

您可以使用 Transit Gateway (TGW) 和 Virtual Private Gateway (VGW) 內建指標來觀察進出 TGW 或 VGW 的網路流量。您可以為這些指標建立警示，以偵測網路流量低於正常層級的情況，指出混合節點和 EKS 控制平面之間可能發生網路問題。下表說明 TGW 和 VGW 指標。

閘道	指標	說明
轉換閘道	BytesIn	TGW 從附件接收的位元組 (EKS 控制平面到混合節點）
轉換閘道	BytesOut	從 TGW 傳送至附件的位元組（混合節點至 EKS 控制平面）
虛擬私有閘道	TunnelDataIn	從連線的 AWS 端透過 VPN 通道傳送到客戶閘道的位元組 (EKS 控制平面到混合節點）
虛擬私有閘道	TunnelDataOut	從客戶閘道透過 VPN 通道在連線的 AWS 端接收的位元組（混合節點到 EKS 控制平面）

您也可以使用 CloudWatch Network Monitor 深入了解混合連線，以減少復原的平均時間，並判斷網路問題是源自 AWS 還是您的環境。CloudWatch Network Monitor 可用來視覺化混合網路連線中的封包遺失和延遲、設定警示和閾值，然後採取行動來改善網路效能。如需詳細資訊，請參閱使用 Amazon CloudWatch Network Monitor。

EKS 提供多種選項來監控叢集和應用程式的運作狀態。對於叢集運作狀態，您可以使用 EKS 主控台中的可觀測性儀表板來快速偵測、疑難排解和修復問題。您也可以使用 Amazon Managed Service for Prometheus、AWS Distro for Open Telemetry (ADOT) 和 CloudWatch 進行叢集、應用程式和基礎設施監控。如需 EKS 可觀測性選項的詳細資訊，請參閱監控叢集效能和檢視日誌。

本機故障診斷

若要準備混合節點和 EKS 控制平面之間的網路連線中斷，您可以設定次要監控和記錄後端，以在區域 AWS 服務無法連線時維持應用程式的可觀測性。例如，您可以設定 AWS Distro for Open Telemetry (ADOT) 收集器，將指標和日誌傳送至多個後端。您也可以使用本機工具，例如 `crictl` CLI，在本機與 Pod 和容器互動，以取代 `kubectl`或其他通常查詢 Kubernetes API 伺服器端點的 Kubernetes API 相容用戶端。如需 的詳細資訊`crictl`，請參閱 cri-tools GitHub 中的 `crictl` 文件。以下列出一些有用的`crictl`命令。

列出在主機上執行的 Pod：

```
crictl pods
```

列出在主機上執行的容器：

```
crictl ps
```

列出在主機上執行的映像：

```
crictl images
```

取得在主機上執行之容器的日誌：

```
crictl logs CONTAINER_NAME
```

取得在主機上執行的 Pod 統計資料：

```
crictl statsp
```

應用程式網路流量

使用混合節點時，請務必考慮並了解應用程式流量的網路流程，以及您用來向叢集外部公開應用程式的技術。不同的應用程式負載平衡和傳入技術在網路連線中斷期間會有不同的行為。例如，如果您使用 Cilium 的 BGP 控制平面功能進行應用程式負載平衡，則 Pod 和服務的 BGP 工作階段可能會在網路連線中斷期間關閉。這是因為 BGP 發言者功能與 Cilium 代理程式整合，並且 Cilium 代理程式會在與 Kubernetes 控制平面中斷連線時持續重新啟動。重新啟動的原因是因為 Cilium 的運作狀態檢查失敗，因為它的運作狀態與 Kubernetes 控制平面的存取權結合（請參閱 CFP：#31702 與 Cilium v1.17 的選擇加入改進）。同樣地，如果您將 Application Load Balancer (ALB) 或 Network Load Balancer (NLB) 用於 AWS 區域來源的應用程式流量，則如果您的現場部署環境失去與 AWS 區域的連線，則該流量可能會暫時中斷。建議您在部署到生產環境之前，先驗證用於負載平衡和傳入的技術在網路連線中斷期間是否保持穩定。aws-samples/eks-hybrid-examples GitHub 儲存庫中的範例使用 MetalLB 在 L2 模式中進行負載平衡，這在混合節點和 EKS 控制平面之間的網路連線中斷期間保持穩定。

檢閱遠端 AWS 服務的相依性

使用混合節點時，請注意您在內部部署或邊緣環境外部的區域 AWS 服務上採取的相依性。範例包括存取 Amazon S3 或 Amazon RDS 以取得應用程式資料、使用 Amazon Managed Service for Prometheus 或 CloudWatch 取得指標和日誌、使用 Application and Network Load Balancer 取得區域來源流量，以及從 Amazon Elastic Container Registry 提取容器。在內部部署環境和 AWS 之間的網路連線中斷期間，將無法存取這些服務。如果您的內部部署環境容易與 AWS 中斷網路連線，請檢閱 AWS 服務的使用情況，並確保失去與這些服務的連線不會危及應用程式的靜態穩定性。

調校 Kubernetes Pod 容錯移轉行為

對於非跨主機可攜式的應用程式，或資源受限的環境，如果沒有 Pod 容錯移轉的備用容量，可以選擇在網路連線中斷期間調整 Pod 容錯移轉行為。一般而言，請務必考慮您應用程式的資源需求，並有足夠的容量讓一或多個應用程式執行個體在節點故障時容錯移轉至不同的主機。

- 選項 1 - 使用 DaemonSets：此選項適用於可以且應該在叢集中所有節點上執行的應用程式。DaemonSets 會自動設定為容忍無法連線的污點，這會讓 DaemonSet Pod 透過網路連線來繫結至其節點。
- 選項 2 - 針對無法連線`tolerationSeconds`的污點進行調校：您可以調校 Pod 在網路中斷期間保持繫結至節點的時間。透過設定應用程式 Pod 來容忍無法連線的污點，在您指定的持續時間 (`tolerationSeconds`在應用程式規格中） 內產生`NoExecute`效果。使用此選項時，當發生網路連線中斷時，您的應用程式 Pod 會持續繫結至節點，直到 `tolerationSeconds` 到期為止。仔細考慮這一點，因為使用 `tolerationSeconds`增加無法連線污點`NoExecute`意味著在無法連線的主機上執行的 Pod 可能需要更長的時間才能移至其他可連線且運作狀態良好的主機。
- 選項 3：自訂控制器：您可以建立並執行自訂控制器（或其他軟體），以監控 Kubernetes 是否有無法連線的污點並產生`NoExecute`效果。偵測到此污點時，自訂控制器可以檢查應用程式特定的指標，以評估應用程式運作狀態。如果應用程式運作狀態良好，自訂控制器可以移除無法連線的污點，防止在網路連線中斷期間從節點移出 Pod。

以下顯示如何使用 `tolerationSeconds`為無法連線的污點設定部署的範例。在此範例中，`tolerationSeconds` 設定為 `1800`(30 分鐘），這表示只有在網路連線中斷持續超過 30 分鐘時，才會移出在無法連線節點上執行的 Pod。

```
apiVersion: apps/v1
kind: Deployment
metadata:
...
spec:
...
      tolerations:
      - key: "node.kubernetes.io/unreachable"
        operator: "Exists"
        effect: "NoExecute"
        tolerationSeconds: 1800
```

透過網路連線的 Kubernetes Pod 容錯移轉

首先，我們會回顧影響 Kubernetes 在節點與 Kubernetes 控制平面之間的網路連線期間行為的主要概念、元件和設定。EKS 符合上游 Kubernetes，因此此處所述的所有 Kubernetes 概念、元件和設定都適用於 EKS 和 EKS 混合節點部署。

EKS 已針對改善網路連線中斷期間的 Pod 容錯移轉行為進行改善，如需詳細資訊，請參閱上游 Kubernetes 儲存庫中的 GitHub 問題 #131294 和 #131481。

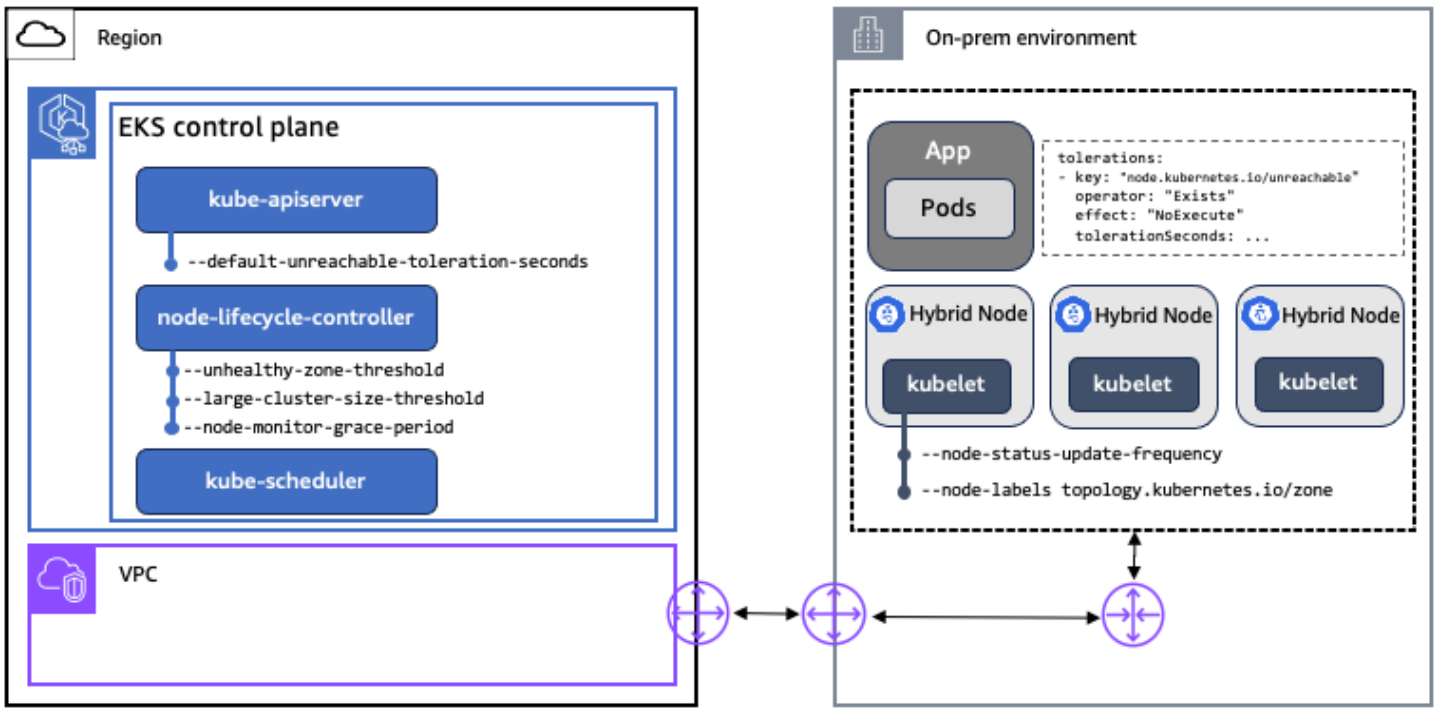
概念

標記和容錯：在 Kubernetes 中使用標記和容錯來控制 Pod 在節點上的排程。污點是由 node-lifecycle-controller 設定，以指出節點不符合排程資格，或應移出這些節點上的 Pod。當節點因網路連線中斷而無法連線時，node-lifecycle-controller會使用 NoSchedule 效果套用 node.kubernetes.io/unreachable 污點，如果符合特定條件，則會套用 NoExecute 效果。node.kubernetes.io/unreachable 污點對應至 NodeCondition 就緒為未知。使用者可以在 PodSpec 中的應用程式層級為污點指定容錯。

- NoSchedule：除非有相符的容錯，否則不會在污點節點上排定新的 Pod。不會移出已在節點上執行的 Pod。
- NoExecute：不會容忍污點的 Pod 會立即被移出。容忍污點的 Pod （ 不指定tolerationSeconds) 會永遠保持繫結。使用指定的tolerationSeconds容忍污點的 Pod 在指定的時間內保持繫結。經過這段時間後，節點生命週期控制器會從節點移出 Pod。

節點租用：Kubernetes 使用租用 API 將 kubelet 節點活動訊號與 Kubernetes API 伺服器通訊。對於每個節點，都有一個具有相符名稱的租用物件。在內部，每個 kubelet 活動訊號都會更新租用物件的 spec.renewTime 欄位。Kubernetes 控制平面會使用此欄位的時間戳記來判斷節點可用性。如果節點與 Kubernetes 控制平面中斷連線，則無法更新其租用的 spec.renewTime，且控制平面會將該節點解釋為 NodeCondition Ready 為未知。

元件



元件	子元件	說明
Kubernetes 控制平面	kube-api-server	API 伺服器是 Kubernetes 控制平面的核心元件，可公開 Kubernetes API。
Kubernetes 控制平面	node-lifecycle-controller	kube-controller-manager 執行的其中一個控制器。它負責偵測和回應節點問題。
Kubernetes 控制平面	kube-scheduler	控制平面元件，可監看沒有指派節點的新建立 Pod，並選取要在其中執行的節點。
Kubernetes 節點	kubelet	在叢集中每個節點上執行的代理程式。kubelet 會監看 PodSpecs，並確保這些 PodSpecs 中所述的容器執行正常。

組態設定

元件	設定	說明	K8s 預設	EKS 預設	可在 EKS 中設定
kube-api-server	default-unreachable-toleration-seconds	指出 的 公差的 tolerationSeconds unreachable:NoExecute ，預設會新增至尚未具有此類公差的每個 Pod。	300	300	否
node-lifecycle-controller	node-monitor-grace-period	節點在標示為運作狀態不佳之前可能沒有回應的時間量。必須是 kubelet 的 N 倍nodeStatusUpdateFrequency ，其中 N 是允許 kubelet 張貼節點狀態的重試次數。	40	40	否
node-lifecycle-controller	large-cluster-size-threshold	對於移出邏輯，node-lifecycle-controller 將叢集視為大型的節點數量。對於此大小或更小的叢集--secondary-node-eviction-rate ，會覆寫為 0。	50	100,000	否
node-lifecycle-controller	unhealthy-zone-threshold	區域中必須尚未就緒，才能將該區域視為運作狀態不佳的節點百分比。	55%	55%	否
kubelet	node-status-update-frequency	kubelet 將節點狀態發佈到控制平面的頻率。必須與 node-lifecycle-controller nodeMonitorGracePeriod 中的 相容。	10	10	是
kubelet	節點標籤	在叢集中註冊節點時要新增的標籤。topology.kubernetes.io/	無	無	是

元件	設定	說明	K8s 預設	EKS 預設	可在 EKS 中設定
		zone 可以使用混合節點指定標籤，將節點分組到區域中。			

透過網路連線的 Kubernetes Pod 容錯移轉

此處所述的行為假設 Pod 以預設設定做為 Kubernetes 部署執行，並使用 EKS 做為 Kubernetes 供應商。實際行為可能會因您的環境、網路連線類型、應用程式、相依性和叢集組態而有所不同。本指南中的內容已使用特定應用程式、叢集組態和外掛程式子集進行驗證。強烈建議在移至生產環境之前，先使用您自己的應用程式來測試行為。

當節點與 Kubernetes 控制平面之間發生網路連線中斷時，每個中斷連線節點上的 kubelet 無法與 Kubernetes 控制平面通訊。因此，在連線還原之前，kubelet 無法移出這些節點上的 Pod。這表示在網路連線中斷之前在這些節點上執行的 Pod 會在中斷連線期間繼續執行，假設沒有其他失敗導致它們關閉。總而言之，您可以在節點與 Kubernetes 控制平面之間的網路連線中斷期間達到靜態穩定性，但在連線還原之前，您無法對節點或工作負載執行變動操作。

有五種主要案例會根據網路連線的性質產生不同的 Pod 容錯移轉行為。在所有情況下，當節點重新連線至 Kubernetes 控制平面時，叢集會再次正常運作，無需操作員介入。以下案例根據我們的觀察概述預期結果，但這些結果可能不適用於所有可能的應用程式和叢集組態。

案例 1：完全叢集中斷

預期結果：無法連線節點上的 Pod 不會移出，並繼續在這些節點上執行。

完整的叢集中斷表示叢集中的所有節點都與 Kubernetes 控制平面中斷連線。在此案例中，控制平面上的node-lifecycle-controller會偵測到叢集中的所有節點都無法連線，並取消任何 Pod 移出。

叢集管理員會在中斷連線Not Ready期間看到狀態為 的所有節點。Pod 狀態不會變更，而且在中斷連線和後續重新連線期間，不會在任何節點上排程新的 Pod。

案例 2：全區域中斷

預期結果：無法連線節點上的 Pod 不會移出，並繼續在這些節點上執行。

完全區域中斷表示該區域中的所有節點都與 Kubernetes 控制平面中斷連線。在此案例中，控制平面上的node-lifecycle-controller會偵測到區域中的所有節點都無法連線，並取消任何 Pod 移出。

叢集管理員會在中斷連線Not Ready期間看到狀態為 的所有節點。Pod 狀態不會變更，而且在中斷連線和後續重新連線期間，不會在任何節點上排程新的 Pod。

案例 3：主要區域中斷

預期結果：無法連線節點上的 Pod 不會移出，並繼續在這些節點上執行。

大多數區域中斷表示指定區域中的大多數節點都與 Kubernetes 控制平面中斷連線。Kubernetes 中的區域由具有相同topology.kubernetes.io/zone標籤的節點定義。如果叢集中未定義任何區域，則大多數中斷表示整個叢集中的大多數節點都會中斷連線。根據預設，大多數是由node-lifecycle-controller的 定義unhealthy-zone-threshold，在 Kubernetes 和 EKS 中均設為 55%。由於 large-cluster-size-threshold EKS 中的 設為 100，000，如果區域中 55% 以上的節點無法連線，則會取消 Pod 移出 （因為大多數叢集遠小於 100，000 個節點）。

叢集管理員會在中斷連線Not Ready期間看到區域中大部分處於 狀態的節點，但 Pod 的狀態不會變更，也不會在其他節點上重新排程這些節點。

請注意，上述行為僅適用於大於三個節點的叢集。在不超過三個節點的叢集中，無法連線節點上的 Pod 會排程為移出，而新的 Pod 會排程在運作狀態良好的節點上。

在測試期間，我們偶爾會觀察到 Pod 在網路連線中斷期間從剛好一個無法連線的節點移出，即使大部分的區域節點都無法連線。作為此行為的原因，我們仍在 Kubernetes node-lifecycle-controller中調查可能的競爭條件。

案例 4：少數區域中斷

預期結果：從無法連線的節點移出 Pod，並在可用的合格節點上排程新的 Pod。

少數中斷表示區域中較少百分比的節點與 Kubernetes 控制平面中斷連線。如果叢集中未定義任何區域，則少數中斷表示整個叢集中節點的少數節點會中斷連線。如前所述，少數是節點node-lifecycle-controllerunhealthy-zone-threshold的設定，預設為 55%。在此案例中，如果網路連線的持續時間超過 default-unreachable-toleration-seconds(5 分鐘） 和 node-monitor-grace-period(40 秒），且區域中少於 55% 的節點無法連線，則新的 Pod 會排程在運作狀態良好的節點上，而無法連線節點上的 Pod 會標記為移出。

叢集管理員將看到在運作狀態良好的節點上建立的新 Pod，中斷連線節點上的 Pod 將顯示為 Terminating。請記住，即使已中斷連線節點上的 Pod 具有 Terminating 狀態，在節點重新連線至 Kubernetes 控制平面之前，它們不會完全移出。

案例 5：網路中斷期間節點重新啟動

預期結果：在節點重新連線至 Kubernetes 控制平面之前，無法連線節點上的 Pod 不會啟動。Pod 容錯移轉遵循案例 1–3 中所述的邏輯，取決於無法連線的節點數量。

網路中斷期間節點重新啟動表示節點上發生另一個故障（例如電源循環、out-of-memory事件或其他問題），同時發生網路中斷連線。如果 kubelet 也已重新啟動，則網路中斷連線開始時在該節點上執行的 Pod 不會在中斷連線期間自動重新啟動。kubelet 會在啟動期間查詢 Kubernetes API 伺服器，以了解應執行哪些 Pod。如果 kubelet 由於網路連線中斷而無法連線到 API 伺服器，則無法擷取啟動 Pod 所需的資訊。

在此案例中，無法使用本機故障診斷工具，例如 `crictl` CLI，將 Pod 手動啟動為「中斷玻璃」量值。Kubernetes 通常會移除故障的 Pod 並建立新的 Pod，而不是重新啟動現有的 Pod（如需詳細資訊，請參閱容器化 GitHub 儲存庫中的 #10213)。靜態 Pod 是唯一由 kubelet 控制的 Kubernetes 工作負載物件，並且可以在這些情況下重新啟動。不過，通常不建議使用靜態 Pod 進行應用程式部署。相反地，在不同的主機之間部署多個複本，以確保在發生多個同時故障時應用程式可用性，例如節點故障，以及節點和 Kubernetes 控制平面之間的網路連線中斷。

透過網路連線的應用程式網路流量

本頁面上的主題與 Kubernetes 叢集聯網和節點與 Kubernetes 控制平面之間的網路連線中斷期間的應用程式流量有關。

Cilium

Cilium 有多種模式可用於 IP 地址管理 (IPAM)、封裝、負載平衡和叢集路由。本指南中驗證的模式使用 Cluster Scope IPAM、VXLAN 浮水印、BGP 負載平衡和 kube-proxy。Cilium 也在沒有 BGP 負載平衡的情況下使用，將其取代為 MetalLB L2 負載平衡。

Cilium 安裝的基礎包含 Cilium 運算子和 Cilium 代理程式。Cilium Operator 會做為部署執行，並註冊 Cilium Custom Resource Definitions (CRDs、管理 IPAM，以及同步叢集物件與 Kubernetes API 伺服器等功能。Cilium 代理程式會以 DaemonSet 的形式在每個節點上執行，並管理 eBPF 程式，以控制叢集上執行工作負載的網路規則。

一般而言，Cilium 設定的叢集內路由在網路連線中斷期間仍然可用且就地，這可以透過觀察 Pod 網路的叢集內流量和 IP 資料表 (iptable) 規則來確認。

```
ip route show table all | grep cilium
```

```
10.86.2.0/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.64/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.128/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.2.192/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16 mtu 1450
10.86.3.0/26 via 10.86.3.16 dev cilium_host proto kernel src 10.86.3.16
10.86.3.16 dev cilium_host proto kernel scope link
...
```

不過，在網路連線中斷期間，Cilium 運算子和 Cilium 代理程式會重新啟動，因為其運作狀態檢查與 Kubernetes API 伺服器連線的運作狀態耦合。預計在網路連線中斷期間，Cilium 運算子和 Cilium 代理程式的日誌中會看到以下內容。在網路連線中斷期間，您可以使用 CLI `crictl` 等工具來觀察這些元件的重新啟動，包括其日誌。

```
msg="Started gops server" address="127.0.0.1:9890" subsys=gops
msg="Establishing connection to apiserver" host="https://<k8s-cluster-ip>:443"
 subsys=k8s-client
msg="Establishing connection to apiserver" host="https://<k8s-cluster-ip>:443"
 subsys=k8s-client
msg="Unable to contact k8s api-server" error="Get \"https://<k8s-cluster-ip>:443/
api/v1/namespaces/kube-system\": dial tcp <k8s-cluster-ip>:443: i/o timeout"
 ipAddr="https://<k8s-cluster-ip>:443" subsys=k8s-client
msg="Start hook failed" function="client.(*compositeClientset).onStart
 (agent.infra.k8s-client)" error="Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/
kube-system\": dial tcp <k8s-cluster-ip>:443: i/o timeout"
msg="Start failed" error="Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/kube-
system\": dial tcp <k8s-cluster-ip>:443: i/o timeout" duration=1m5.003834026s
msg=Stopping
msg="Stopped gops server" address="127.0.0.1:9890" subsys=gops
msg="failed to start: Get \"https://<k8s-cluster-ip>:443/api/v1/namespaces/kube-system
\": dial tcp <k8s-cluster-ip>:443: i/o timeout" subsys=daemon
```

如果您使用 Cilium 的 BGP 控制平面功能進行應用程式負載平衡，則 Pod 和服務的 BGP 工作階段可能會在網路中斷連線期間關閉，因為 BGP 發言者功能已與 Cilium 代理程式整合，且 Cilium 代理程式會在中斷與 Kubernetes 控制平面的連線時持續重新啟動。如需詳細資訊，請參閱 Cilium 文件中的 Cilium BGP 控制平面操作指南。此外，如果您在網路中斷期間同時遇到故障，例如電源循環或機器重新開機，則不會透過這些動作保留 Cilium 路由，不過當節點重新連線至 Kubernetes 控制平面且 Cilium 再次啟動時，會重新建立路由。

Calico

即將推出

MetalLB

MetalLB 有兩種負載平衡模式：L2 模式和 BGP 模式。如需這些負載平衡模式的運作方式及其限制的詳細資訊，請參閱 MetalLB 文件。本指南的驗證在 L2 模式中使用 MetalLB，其中叢集中的一台機器取得 Kubernetes Service 的所有權，並使用 ARP for IPv4 讓負載平衡器 IP 地址可在本機網路上連線。執行 MetalLB 時，有一個控制器負責在每個節點上執行的 IP 指派和發言者，這些節點負責使用指派的 IP 地址公告服務。MetalLB 控制器會做為部署執行，而 MetalLB 發言者則會做為 DaemonSet 執行。在網路連線中斷期間，MetalLB 控制器和發言者無法監看叢集資源的 Kubernetes API 伺服器，但繼續執行。最重要的是，在網路連線中斷期間，使用 MetalLB 進行外部連線的 服務仍然可用且可存取。

kube-proxy

在 EKS 叢集中，kube-proxy 在每個節點上以 DaemonSet 的形式執行，並負責管理網路規則，透過將服務 IP 地址轉譯為基礎 Pod 的 IP 地址來啟用服務和 Pod 之間的通訊。由 kube-proxy 設定的 IP 資料表 (iptable) 規則會在網路連線中斷期間維護，叢集內路由會繼續運作，而 kube-proxy Pod 會繼續執行。

您可以使用下列 iptables 命令來觀察 kube-proxy 規則。第一個命令顯示經過PREROUTING鏈結的封包會導向至KUBE-SERVICES鏈結。

```
iptables -t nat -L PREROUTING
```

```
Chain PREROUTING (policy ACCEPT)
target         prot opt source      destination
KUBE-SERVICES  all  --  anywhere    anywhere      /* kubernetes service portals */
```

檢查KUBE-SERVICES鏈結，我們可以看到各種叢集服務的規則。

```
Chain KUBE-SERVICES (2 references)
target                     prot opt source      destination
KUBE-SVL-NZTS37XDTDNXGCKJ  tcp  --  anywhere    172.16.189.136  /* kube-system/hubble-
peer:peer-service cluster IP /
KUBE-SVC-2BINP2AXJOTI3HJ5  tcp  --  anywhere    172.16.62.72    / default/metallb-
webhook-service cluster IP /
KUBE-SVC-LRNEBRA3Z5YGJ4QC  tcp  --  anywhere    172.16.145.111  / default/redis-leader
 cluster IP /
KUBE-SVC-I7SKRZYQ7PWYV5X7  tcp  --  anywhere    172.16.142.147  / kube-system/eks-
extension-metrics-api:metrics-api cluster IP /
KUBE-SVC-JD5MR3NA4I4DYORP  tcp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:metrics cluster IP /
```

```
KUBE-SVC-TCOU7JCQXEZGVUNU  udp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:dns cluster IP /
KUBE-SVC-ERIFXISQEP7F7OF4  tcp  --  anywhere    172.16.0.10     / kube-system/kube-
dns:dns-tcp cluster IP /
KUBE-SVC-ENODL3HWJ5BZY56Q  tcp  --  anywhere    172.16.7.26     / default/frontend
 cluster IP /
KUBE-EXT-ENODL3HWJ5BZY56Q  tcp  --  anywhere    <LB-IP>    / default/frontend
 loadbalancer IP /
KUBE-SVC-NPX46M4PTMTKRN6Y  tcp  --  anywhere    172.16.0.1      / default/
kubernetes:https cluster IP /
KUBE-SVC-YU5RV2YQWHLZ5XPR  tcp  --  anywhere    172.16.228.76   / default/redis-
follower cluster IP /
KUBE-NODEPORTS             all  --  anywhere    anywhere        / kubernetes service
 nodeports; NOTE: this must be the last rule in this chain */
```

檢查前端服務的鏈結是否有應用程式，我們可以看到支援服務的 Pod IP 地址。

```
iptables -t nat -L KUBE-SVC-ENODL3HWJ5BZY56Q
```

```
Chain KUBE-SVC-ENODL3HWJ5BZY56Q (2 references)
target                     prot opt source    destination
KUBE-SEP-EKXE7ASH7Y74BGBO  all  --  anywhere  anywhere    /* default/frontend ->
 10.86.2.103:80 / statistic mode random probability 0.33333333349
KUBE-SEP-GCY3OUXWSVMSEAR6  all  --  anywhere  anywhere    / default/frontend ->
 10.86.2.179:80 / statistic mode random probability 0.50000000000
KUBE-SEP-6GJJR3EF5AUP2WBU  all  --  anywhere  anywhere    / default/frontend ->
 10.86.3.47:80 */
```

下列 kube-proxy 日誌訊息預期會在網路連線中斷期間嘗試監看 Kubernetes API 伺服器，以取得節點和端點資源的更新。

```
"Unhandled Error" err="k8s.io/client-go/informers/factory.go:160: Failed to watch
 *v1.Node: failed to list *v1.Node: Get \"https://<k8s-endpoint>/api/v1/nodes?
fieldSelector=metadata.name%3D<node-name>&resourceVersion=2241908\": dial tcp <k8s-
ip>:443: i/o timeout" logger="UnhandledError"
"Unhandled Error" err="k8s.io/client-go/informers/factory.go:160: Failed to watch
 *v1.EndpointSlice: failed to list *v1.EndpointSlice: Get \"https://<k8s-endpoint>/
apis/discovery.k8s.io/v1/endpointslices?labelSelector=%21service.kubernetes.io
%2Fheadless%2C%21service.kubernetes.io%2Fservice-proxy-name&resourceVersion=2242090\":
 dial tcp <k8s-ip>:443: i/o timeout" logger="UnhandledError"
```

CoreDNS

根據預設，EKS 叢集中的 Pod 會使用 CoreDNS 叢集 IP 地址做為叢集內 DNS 查詢的名稱伺服器。在 EKS 叢集中，CoreDNS 會在節點上執行為部署。使用混合節點時，當混合節點上有在本機執行的 CoreDNS 複本時，Pod 可以在網路連線中斷期間繼續與 CoreDNS 通訊。如果您的 EKS 叢集的雲端節點和內部部署環境中的混合節點，建議您在每個環境中至少有一個 CoreDNS 複本。CoreDNS 會繼續為在網路中斷連線之前建立的記錄提供 DNS 查詢，並繼續透過網路重新連線執行，以確保靜態穩定性。

下列 CoreDNS 日誌訊息預期會在網路中斷連線期間從 Kubernetes API 伺服器列出物件。

```
Failed to watch *v1.Namespace: failed to list *v1.Namespace: Get "https://<k8s-cluster-
ip>:443/api/v1/namespaces?resourceVersion=2263964": dial tcp <k8s-cluster-ip>:443: i/o
 timeout
Failed to watch *v1.Service: failed to list *v1.Service: Get "https://<k8s-cluster-
ip>:443/api/v1/services?resourceVersion=2263966": dial tcp <k8s-cluster-ip>:443: i/o
 timeout
Failed to watch *v1.EndpointSlice: failed to list *v1.EndpointSlice: Get "https://<k8s-
cluster-ip>:443/apis/discovery.k8s.io/v1/endpointslices?resourceVersion=2263896": dial
 tcp <k8s-cluster-ip>: i/o timeout
```

透過網路連線的主機登入資料

EKS 混合節點與 AWS Systems Manager (SSM) 混合啟用和 AWS IAM Roles Anywhere 整合，用於使用 EKS 控制平面驗證節點的臨時 IAM 登入資料。SSM 和 IAM Roles Anywhere 都會自動重新整理他們在內部部署主機上管理的臨時憑證。建議您在叢集中跨混合節點使用單一登入資料提供者，無論是 SSM 混合啟用或 IAM Roles Anywhere，但不能同時使用兩者。

SSM 混合啟用

SSM 佈建的臨時登入資料有效期為一小時。使用 SSM 做為登入資料提供者時，您無法變更登入資料有效期間。SSM 會在臨時登入資料過期之前自動輪換，輪換不會影響節點或應用程式的狀態。不過，當 SSM 代理程式與 SSM 區域端點之間發生網路連線中斷時，SSM 無法重新整理登入資料，登入資料可能會過期。

如果 SSM 無法連線至 SSM 區域端點，SSM 會使用指數退避進行登入資料重新整理重試。在 SSM 代理程式版本 `3.3.808.0` 和更新版本 (2024 年 8 月發行） 中，指數退避上限為 30 分鐘。視網路連線的持續時間而定，SSM 重新整理登入資料最多可能需要 30 分鐘，而且在重新整理登入資料之前，混合節點不會重新連線至 EKS 控制平面。在此案例中，您可以重新啟動 SSM 代理程式來強制重新整理登入資料。作為目前 SSM 憑證重新整理行為的副作用，節點可能會在不同時間重新連線，具體取決於

每個節點上的 SSM 代理程式何時設法重新整理其憑證。因此，您可能會看到來自尚未重新連線至已重新連線節點之節點的 Pod 容錯移轉。

取得 SSM 代理程式版本。您也可以查看 SSM 主控台的機群管理員區段：

```
# AL2023, RHEL
yum info amazon-ssm-agent
# Ubuntu
snap list amazon-ssm-agent
```

重新啟動 SSM 代理程式：

```
# AL2023, RHEL
systemctl restart amazon-ssm-agent
# Ubuntu
systemctl restart snap.amazon-ssm-agent.amazon-ssm-agent
```

檢視 SSM 代理程式日誌：

```
tail -f /var/log/amazon/ssm/amazon-ssm-agent.log
```

網路連線中斷期間的預期日誌訊息：

```
INFO [CredentialRefresher] Credentials ready
INFO [CredentialRefresher] Next credential rotation will be in 29.995040663666668
 minutes
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 35s before retrying retrieve credentials
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 56s before retrying retrieve credentials
ERROR [CredentialRefresher] Retrieve credentials produced error: RequestError: send
 request failed
INFO [CredentialRefresher] Sleeping for 1m24s before retrying retrieve credentials
```

IAM Roles Anywhere

根據預設，IAM Roles Anywhere 佈建的臨時登入資料有效期為一小時。您可以透過 IAM Roles Anywhere 設定檔中的 durationSeconds 欄位，使用 IAM Roles Anywhere 設定憑證有效期間。登

入資料有效性的持續時間上限為 12 小時。混合節點 IAM 角色上的`MaxSessionDuration`設定必須大於 IAM Roles Anywhere 設定檔上的`durationSeconds`設定。

使用 IAM Roles Anywhere 做為混合節點的登入資料提供者時，網路中斷後通常會在網路還原後的幾秒鐘內重新連線至 EKS 控制平面，因為 kubelet 會呼叫 `aws_signing_helper credential-process` 來隨需取得登入資料。雖然與混合節點或網路連線不直接相關，但您可以在使用 IAM Roles Anywhere 時設定憑證過期的通知和提醒。如需詳細資訊，請參閱在 IAM Roles Anywhere 中自訂通知設定。

執行 AI/ML 工作負載的最佳實務

> **Tip**
>
> 註冊即將舉行的 Amazon EKS AI/ML 研討會。

在 EKS 上執行 AI/ML 工作負載時實作最佳實務，可確保這些工作負載具有效能、成本效益、彈性和適當資源。最佳實務分為下列一般區段：運算、聯網、儲存、可觀測性和效能。

意見回饋

本指南已在 GitHub 上發佈，以便收集更廣泛的 EKS/Kubernetes 社群的直接意見回饋和建議。如果您有最佳實務，認為我們應該包含在指南中，請在 GitHub 儲存庫中提出問題或提交 PR。我們的目的是在新功能新增至服務或新的最佳實務演進時，定期更新指南。

運算和自動擴展

> **Tip**
>
> 註冊即將舉行的 Amazon EKS AI/ML 研討會。

GPU 資源最佳化和成本管理

使用 Well-Known 標籤以 GPU 需求排程工作負載

對於對不同 GPU 特性（例如 GPU、GPU 記憶體）敏感的 AI/ML 工作負載，建議使用與 Karpenter 和受管節點群組搭配使用的節點類型支援的已知排程標籤來指定 GPU 需求。如果未定義這些項目，可能會導致在 GPU 資源不足的執行個體上排程 Pod，進而導致故障或效能降低。建議使用 nodeSelector 或 Node 親和性來指定 Pod 應在哪個節點上執行，並在 Pod 的資源區段中設定運算資源 (CPU、記憶體、GPUs 等）。

範例

例如，使用 Karpenter 時使用 GPU 名稱節點選擇器：

```
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod-example
spec:
  containers:
  - name: ml-workload
    image: <image>
    resources:
      limits:
        nvidia.com/gpu: 1  # Request one NVIDIA GPU
  nodeSelector:
    karpenter.k8s.aws/instance-gpu-name: "l40s"  # Run on nodes with NVIDIA L40S GPUs
```

使用 Kubernetes 裝置外掛程式公開 GPUs

若要在節點上公開 GPUs，NVIDIA GPU 驅動程式必須安裝在節點的作業系統和容器執行時間上，設定成允許 Kubernetes 排程器將 Pod 指派給具有可用 GPUs 的節點。NVIDIA Kubernetes 裝置外掛程式的設定程序取決於您使用的 EKS 加速 AMI：

- Bottlerocket Accelerated AMI：此 AMI 包含 NVIDIA GPU 驅動程式，且已預先安裝 NVIDIA Kubernetes 裝置外掛程式並可供使用，可立即支援 GPU。向 Kubernetes 排程器公開 GPUs 不需要額外的組態。
- AL2023 加速 AMI：此 AMI 包含 NVIDIA GPU 驅動程式，但未預先安裝 NVIDIA Kubernetes 裝置外掛程式。您必須分別安裝和設定裝置外掛程式，通常是透過 DaemonSet。請注意，如果您使用 eksctl 建立叢集，並在 ClusterConfig 中指定 GPU 執行個體類型（例如 `g5.xlarge`），`eksctl` 會自動選取適當的 AMI 並安裝 NVIDIA Kubernetes 裝置外掛程式。若要進一步了解，請參閱 eksctl 文件中的 GPU 支援。

如果您決定改用 EKS 加速 AMIs和 NVIDIA GPU 運算子來管理 NVIDIA Kubernetes 裝置外掛程式等元件，請注意，根據預先安裝的 NVIDIA GPU 驅動程式和 NVIDIA 容器工具組 NVIDIA 文件，停用 NVIDIA GPU 驅動程式和 NVIDIA 容器工具組的管理。

若要驗證 NVIDIA 裝置外掛程式是否作用中且 GPUs 已正確公開，請執行：

```
kubectl describe node | grep nvidia.com/gpu
```

此命令會檢查`nvidia.com/gpu`資源是否在節點的容量和可配置的資源中。例如，具有一個 GPU 的節點應該會顯示 `nvidia.com/gpu: 1`。如需詳細資訊，請參閱 Kubernetes GPU 排程指南。

使用許多不同的 EC2 執行個體類型

盡可能使用多種不同的 EC2 執行個體類型，是 Amazon EKS 上可擴展性的重要最佳實務，如 the section called “資料平面”節所述。此建議也適用於具有加速硬體（例如 GPUs) 的執行個體。如果您建立的叢集僅使用一種執行個體類型，並嘗試擴展超過該區域的容量的節點數量，您可能會收到容量不足錯誤 (ICE)，表示沒有可用的執行個體。在任意多樣化之前，請務必了解 AI/ML 工作負載的獨特特性。使用 EC2 Instance Type Explorer 工具檢閱可用的執行個體類型，以產生符合您特定運算需求的執行個體類型清單，並避免任意限制可在叢集中使用的執行個體類型。

加速運算執行個體提供不同的購買模型，以符合短期、中期和穩定狀態工作負載。對於短期、靈活且容錯的工作負載，如果您想要避免進行保留，請查看 Spot 執行個體。容量區塊、隨需執行個體和 Save Plans 可讓您佈建中長期工作負載持續時間的加速運算執行個體。為了提高成功存取您偏好購買選項中所需容量的機會，建議使用各種執行個體類型和可用區域清單。或者，如果您遇到特定購買模型 ICEs，請使用不同的模型重試。

範例 以下範例示範如何啟用 Karpenter NodePool 來佈建大於第 3 代的 G 和 P 執行個體（例如 p3)。如需進一步了解，請參閱可擴展性區段。

```
- key: karpenter.k8s.aws/instance-category
  operator: In
  values: ["g", "p"] # Diversifies across G-series and P-series
- key: karpenter.k8s.aws/instance-generation
  operator: Gt
  values: ["3"] # Selects instance generations greater than 3
```

如需使用適用於 GPUs 的 Spot 執行個體的詳細資訊，請參閱以下「考慮使用適用於 GPUs 的 Amazon EC2 Spot 執行個體搭配 Karpenter」。

考慮將 Amazon EC2 Spot 執行個體與 Karpenter 搭配使用於 GPUs

Amazon EC2 Spot 執行個體可讓您利用 AWS 雲端中未使用的 EC2 容量，相較於隨需價格，提供高達 90% 的折扣。當 Amazon EC2 需要恢復容量時，Amazon EC2 Spot 執行個體可以透過兩分鐘的通知中斷。如需詳細資訊，請參閱《Amazon EC2 使用者指南》中的 Spot 執行個體。Amazon EC2 Spot 是容錯、無狀態和彈性（時間和執行個體類型）工作負載的理想選擇。若要進一步了解何時使用 Spot 執行個體，請參閱 EC2 Spot 執行個體最佳實務。如果適用於 Spot 的 AI/ML 工作負載，您也可以使用 Spot 執行個體。

使用案例

適合 Spot 的工作負載可以是大數據、容器化工作負載、CI/CD、無狀態 Web 伺服器、高效能運算 (HPC) 和轉譯工作負載。Spot 執行個體不適用於無彈性、具狀態、容錯能力或執行個體節點之間緊密

耦合的工作負載（例如，具有彼此高度依賴的平行程序以進行運算的工作負載，需要持續的節點間通訊，例如以 MPI 為基礎的高效能運算應用程式，例如運算流體動態或具有複雜相互依存性的分散式資料庫）。以下是我們建議的特定使用案例（無特定順序）：

- 即時線上推論：只要您的工作負載易於使用 Spot 執行個體，即可針對即時推論工作負載使用 Spot 執行個體進行成本最佳化擴展。換句話說，推論時間可能不到兩分鐘，應用程式對中斷具有容錯能力，並且可以在不同的執行個體類型上執行。透過執行個體多樣性（例如跨多個執行個體類型和可用區域）或保留來確保高可用性，同時實作應用程式層級容錯能力來處理潛在的 Spot 中斷。
- 超參數調校：使用 Spot 執行個體以機會方式執行探索性調校任務，因為可以容忍中斷而不會造成重大損失，尤其是短期實驗。
- 資料擴增：使用 Spot 執行個體執行資料預先處理和擴增任務，以便在中斷時從檢查點重新啟動，使其成為 Spot 變數可用性的理想選擇。
- 微調模型：使用 Spot 執行個體進行微調，搭配強大的檢查點機制，從上次儲存的狀態恢復，將執行個體中斷的影響降至最低。
- 批次推論：使用 Spot 執行個體以non-real-time的方式處理大量離線推論請求，其中任務可以暫停和恢復，提供與 Spot 節省成本的最佳一致性，並透過重試或多樣化處理潛在的中斷。
- 機會性訓練子集：將 Spot 執行個體用於邊緣或實驗訓練工作負載（例如，小於 1，000 萬個參數的較小模型），其中可接受中斷，並可套用跨執行個體類型或區域多樣化等效率最佳化，但由於潛在中斷，不建議用於生產規模訓練。

考量

若要在 Amazon EKS 上將 Spot 執行個體用於加速工作負載，有許多重要考量（無特定順序）：

- 使用 Karpenter 來管理已啟用進階整合的 Spot 執行個體。透過在 Karpenter NodePool 中將 karpenter.sh/capacity-type 指定為「spot」，Karpenter 預設會佈建 Spot 執行個體，而不需要任何額外的組態。不過，若要啟用進階 Spot-to-Spot 整合，以低價 Spot 替代方案取代未充分利用的 Spot 節點，您需要透過設定 --feature-gates SpotToSpotConsolidation=true in Karpenter 控制器引數或透過 FEATURE_GATES 環境變數來啟用 SpotToSpotConsolidation 功能閘道。Karpenter price-capacity-optimized配置策略來佈建 EC2 執行個體。根據 NodePool 要求和 Pod 限制，Karpenter bin-packs 無法排程的 Pod，並將各種執行個體類型傳送至 Amazon EC2 機群 API。您可以使用 EC2 Instance Type Explorer 工具來產生符合您特定運算需求的執行個體類型清單。
- 確保工作負載無狀態、容錯能力和彈性。工作負載在執行個體/GPU 大小方面必須無狀態、容錯且靈活。這可在 Spot 中斷後無縫恢復，而執行個體彈性可讓您在 Spot 上停留更長的時間。使用 AWS SQS 佇列的名稱設定 settings.interruptionQueue Helm 值，以擷取 Spot 中斷事件，在 Karpenter

中啟用 Spot 中斷處理。例如，透過 Helm 安裝 時，請使用 --set "settings.interruptionQueue=${CLUSTER_NAME}"。若要查看範例，請參閱 Karpenter 入門指南。當 Karpenter 注意到 Spot 中斷事件時，它會在中斷事件之前自動封鎖、污點、耗盡和終止節點（以最大化 Pod 的終止寬限期）。同時，Karpenter 會立即啟動新的節點，以便盡快準備就緒。

- 避免過度限制執行個體類型選擇。您應該盡可能避免限制執行個體類型。透過不限制執行個體類型，以較低的成本取得大規模 Spot 容量，並以較低的 Spot 執行個體中斷頻率取得的機會較高。例如，避免限制特定類型（例如 g5.xlarge)。請考慮使用 karpenter.k8s.aws/instance-category 和 karpenter.k8s.aws/instance-generation 等金鑰來指定各種執行個體類別和世代。Karpenter 可簡化跨多個執行個體類型和可用區域 (AZs) 的隨需和 Spot 執行個體容量多樣化。此外，如果您的 AI/ML 工作負載需要特定或有限數量的加速器，但區域之間具有彈性，您可以使用 Spot Placement Score 在啟動之前動態識別部署工作負載的最佳區域。
- 擴展 NodePool 需求，以包含更多類似的 EC2 執行個體系列。每個 Spot 執行個體集區都包含特定可用區域 (AZ) 中特定執行個體類型的未使用 EC2 執行個體容量。當 Karpenter 嘗試佈建新節點時，會選取符合 NodePool 需求的執行個體類型。如果任何 AZ 中沒有相容的執行個體類型具有 Spot 容量，則佈建會失敗。為了避免此問題，請允許跨大小和可用區域 (AZs) 的 NVIDIA 更廣泛的 g 系列執行個體（第 4 代或更高版本），同時考慮 GPU 記憶體或 Ray Tracing 等硬體需求。由於執行個體的類型可能不同，因此您需要確保您的工作負載能夠在每種類型上執行，以及您取得的效能符合您的需求。
- 利用區域中的所有可用區域。可用容量因可用區域 (AZ) 而異，特定執行個體類型可能無法在一個可用區域中使用，但在另一個可用區域中則有許多。執行個體類型和可用區域的每個唯一組合都會構成個別的 Spot 容量集區。透過請求 Karpenter NodePool 要求區域內所有AZs容量，您可以一次有效地搜尋更多集區。這可將 Spot 容量集區的數量最大化，因此會增加取得 Spot 容量的機率。若要達成此目的，請在 NodePool 組態中完全省略 topology.kubernetes.io/zone 金鑰，以允許 Karpenter 從區域中所有可用的可用 AZs 中選取，或使用運算子明確列出 AZs：在 中，並提供值（例如 us-west-2a)。
- 考慮使用 Spot 配置分數 (SPS) 來了解使用 Spot 執行個體成功存取所需容量的可能性。Spot 配置分數 (SPS) 是一種工具，可提供分數，協助您評估 Spot 請求成功的可能性。當您使用 SPS 時，首先指定 Spot 執行個體的運算需求，然後 Amazon EC2 會傳回 Spot 請求可能成功的前 10 個區域或可用區域 (AZs)。區域和可用區域的分數範圍是從 1 到 10。分數為 10 表示您的 Spot 請求很有可能，但不保證成功。分數 1 表示您的 Spot 請求完全不可能成功。對於不同的區域或可用區域，可能會傳回相同的分數。若要進一步了解，請參閱在 AWS 上建立 Spot 配置分數追蹤器儀表板的指引。隨著 Spot 容量隨時波動，SPS 將協助您識別最適合工作負載限制（即彈性、效能、大小等）AZs區域和區域的組合。如果您的 AI/ML 工作負載需要特定或有限數量的加速器，但區域之間具有彈性，您可以使用 Spot 置放分數，在啟動之前動態識別部署工作負載的最佳區域。為了協助您自動找出取得 Spot 容量的可能性，我們提供建置 SPS 追蹤器儀表板的指引。此解決方案使用 YAML 組態進行多樣化的設定（例如，包括 GPUs 的執行個體需求）、在 CloudWatch 中存放指標，並提供儀表板

來比較組態，以監控一段時間的 SPS 分數。定義每個工作負載的儀表板，以評估 vCPU、記憶體和 GPU 需求，確保 EKS 叢集的最佳設定，包括考慮使用其他 AWS 區域。若要進一步了解，請參閱 Spot 配置分數的運作方式。

- 優雅地處理 Spot 中斷和測試。對於終止期間超過兩分鐘的 Pod，舊節點會在重新排程這些 Pod 之前中斷，這可能會影響工作負載可用性。設計應用程式時，請考慮兩分鐘的 Spot 中斷通知、在長時間執行的應用程式中實作檢查點（例如，將進度儲存至 Amazon S3 等持久性儲存體）以在中斷後繼續、在 Pod 規格中延長 terminationGracePeriodSeconds（預設為 30 秒），以允許更多時間進行正常關機，以及在應用程式中使用preStop生命週期勾點和/或 SIGTERM 訊號處理中斷，以進行正常關機活動，例如清除、狀態儲存和連線關閉。對於即時工作負載，擴展時間很重要，且工作負載需要超過兩分鐘的時間才能準備好為流量提供服務，請考慮透過檢閱 和the section called “效能”最佳實務來最佳化容器啟動the section called “儲存”和 ML 模型載入時間。若要測試替代節點，請使用 AWS Fault Injection Service (FIS) 來模擬 Spot 中斷。

除了這些核心 Spot 最佳實務之外，在 Amazon EKS 上管理 GPU 工作負載時，請考量這些因素。與 CPU 工作負載不同，GPU 工作負載對 GPU 功能和可用 GPU 記憶體等硬體詳細資訊特別敏感。GPU 工作負載可能受限於他們可以使用的執行個體類型，與 CPUs 相比，可用的選項較少。首先，評估工作負載是否具有執行個體彈性。如果您不知道工作負載可以使用多少執行個體類型，請個別進行測試，以確保相容性和功能。識別您可以多靈活地盡可能多樣化，同時確認多樣化可讓工作負載正常運作並了解任何效能影響（例如輸送量或完成時間）。在多樣化工作負載的過程中，請考慮下列事項：

- 檢閱 CUDA 和架構相容性。您的 GPU 工作負載可能針對特定硬體、GPU 類型（例如，p3 中的 V100 和 p4 中的 A100) 進行最佳化，或針對 TensorFlow 等程式庫的特定 CUDA 版本進行寫入，因此請務必檢閱工作負載的相容性。此相容性對於防止 GPU 加速中的執行時間錯誤、當機、故障（例如，與 PyTorch 或 TensorFlow 等架構不相符的 CUDA 版本可能會阻止執行）或利用 FP16/INT8 精確度等硬體功能至關重要。
- GPU 記憶體。請務必評估模型的記憶體需求，並使用 DCGM Exporter 等工具在執行時間分析模型的記憶體用量，並在 karpenter.k8s.aws/instance-gpu-memory 等知名標籤中設定執行個體類型所需的最低 GPU 記憶體。GPU VRAM 因執行個體類型而異（例如，NVIDIA T4 具有 16GB、A10G 具有 24GB、V100 具有 16-32GB)，而 ML 模型（例如，大型語言模型）可能會超過可用的記憶體，導致out-of-memory(OOM) 錯誤或當機。對於 EKS 中的 Spot 執行個體，這可能會限制多樣化。例如，如果您的模型不適合，則不能包含較低的 VRAM 類型，這可能會限制對容量集區的存取並提高中斷風險。請注意，對於單一 GPU、單一節點推論（例如，在同一節點上排程使用其 GPU 資源的多個 Pod)，這可能會限制多樣化，因為您只能在 Spot 組態中包含具有足夠 VRAM 的執行個體類型。

- 浮點精確度和效能。並非所有 Nvidia GPU 架構都有相同的浮點精確度（例如 FP16/INT8)。評估工作負載所需的核心類型 (CUDA/Tensor/RT) 效能和浮點精確度。在價格較低、效能較低的 GPU 上執行並不表示效能更好，因此請考慮在特定時間範圍內完成的工作評估效能，以了解多樣化的影響。

案例：即時推論工作負載的多樣化

對於 Spot 執行個體上的即時線上推論工作負載，您可以設定 Karpenter NodePool 以分散於相容的 GPU 執行個體系列和世代。這種方法透過從多個 Spot 集區提取來確保高可用性，同時透過 GPU 功能、記憶體和架構的限制來維持效能。它支援在限制執行個體容量時使用替代方案，將中斷降至最低，並最佳化推論延遲。此範例 NodePool 狀態使用大於 3 的 g 和 p 系列執行個體，這些執行個體具有超過 20GB 的 GPU 記憶體。

範例

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu-inference-spot
spec:
  template:
    metadata:
      labels:
        role: gpu-spot-worker
    spec:
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["spot"] # Use Spot Instances
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["g", "p"] # Diversifies across G-series and P-series
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["3"] # Selects instance generations greater than 3
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"] # Specifies AMD64 architecture, compatible with NVIDIA GPUs
        - key: karpenter.k8s.aws/instance-gpu-memory
          operator: Gt
          values: ["20480"] # Ensures more than 20GB (20480 MiB) total GPU memory
      taints:
        - key: nvidia.com/gpu
```

```
          effect: NoSchedule
      nodeClassRef:
        name: gpu-inference-ec2
        group: karpenter.k8s.aws
        kind: EC2NodeClass
      expireAfter: 720h
  limits:
    cpu: 100
    memory: 100Gi
  disruption:
    consolidationPolicy: WhenEmptyOrUnderutilized
    consolidateAfter: 5m # Enables consolidation of underutilized nodes after 5 minutes
```

實作長時間執行訓練任務的檢查點

檢查點是一種容錯技術，涉及定期儲存程序的狀態，以便在中斷時從上次儲存的點恢復。在機器學習中，它通常與訓練相關，其中長時間執行的任務可以節省模型權重和最佳化工具狀態，以便在失敗後繼續訓練，例如硬體問題或 Spot 執行個體中斷。

您可以在訓練期間使用檢查點來儲存機器學習 (ML) 模型的狀態。檢查點是模型的快照，並且可以透過機器學習 (ML) 架構的回呼函式進行設定。您可以使用儲存的檢查點，從上次儲存的檢查點重新啟動訓練任務。使用檢查點，您可以儲存訓練中的模型快照，因為訓練任務或執行個體意外中斷。這可讓您在未來從檢查點繼續訓練模型。除了實作節點彈性系統之外，建議您實作檢查點，以減輕中斷的影響，包括硬體故障或 Amazon EC2 Spot 執行個體中斷所造成的中斷。

如果沒有檢查點，中斷可能會導致運算時間浪費和進度遺失，這對於長時間執行的訓練任務非常昂貴。檢查點可讓任務定期儲存其狀態（例如模型權重和最佳化工具狀態），並在中斷後從最後一個檢查點（上次處理的批次）恢復。若要實作檢查點，請設計您的應用程式以處理大量資料，並在訓練任務進行時透過 Amazon S3 CSI 驅動程式掛載點將中繼結果儲存到持久性儲存體，例如 Amazon S3 儲存貯體。

使用案例

檢查點在特定案例中特別有用，以平衡容錯能力與效能額外負荷。在下列情況下，請考慮使用檢查點：

- 任務持續時間超過幾個小時：對於長時間執行的訓練任務（例如，對於小型模型 >1-2 小時，對於具有數十億個參數的大型基礎模型，則為天數/週），其中中斷造成的進度損失成本很高。較短的任務可能不會證明 I/O 額外負荷。
- 對於 Spot 執行個體或硬體故障：在容易中斷的環境中，例如 EC2 Spot (2 分鐘通知）或硬體故障（例如 GPU 記憶體錯誤），檢查點可快速恢復，讓 Spot 能夠節省容錯工作負載的成本。

- 大規模分散式訓練：適用於具有數百/數千個加速器的設定（例如 >100 GPUs)，其中故障之間的平均時間會隨規模線性減少。使用 進行模型/資料平行處理，以處理並行檢查點存取並避免完全重新啟動。
- 具有高資源需求的大規模模型：在 PB 級 LLM 訓練中，失敗是因叢集大小而不可避免的；分層方法（暫時性方法為每 5-30 分鐘快速本機一次，主要故障為每小時耐用一次）可最佳化復原時間與效率。

使用 ML 容量區塊來確保 P 和 Trainium 執行個體的容量

ML 的容量區塊可讓您保留熱門的 GPU 執行個體，特別是 P 執行個體（例如 p6-b200、p5、p5e、p5en、p4d、p4de) 和 Trainium 執行個體（例如 trn1、trn2)，以幾乎立即開始或在未來日期開始，以支援您的短期機器學習 (ML) 工作負載。這些保留非常適合用來確保運算密集型任務的容量，例如模型訓練和微調。EC2 容量區塊定價包含保留費和作業系統費用。若要進一步了解定價，請參閱適用於 ML 定價的 EC2 容量區塊。

若要為 Amazon EKS 上的 AI/ML 工作負載保留 GPUs 以保證可預測的容量，我們建議將 ML 容量區塊用於短期或隨需容量保留 ODCRs)，以保證一般用途的容量。

- ODCRs 可讓您在特定可用區域中保留 EC2 執行個體容量（例如 g5 或 p5 等 GPU 執行個體），即使在高需求期間也能確保可用性。ODCRs沒有長期承諾，但您要支付預留容量的隨需費率，無論是已使用或閒置。在 EKS 中，Karpenter 和受管節點群組等節點類型支援 ODCRs。若要排定 Karpenter 中的 ODCRs優先順序，請將 NodeClass 設定為使用 `capacityReservationSelectorTerms` 欄位。請參閱 Karpenter NodePools 文件。
- 容量區塊是 GPU（例如 p5、p4d) 或 Trainium (trn1、trn2) 執行個體的特殊保留機制，專為模型訓練、微調或實驗等短期 ML 工作負載而設計。您從未來日期開始，在定義的期間（通常為 24 小時至 182 天）保留容量，僅支付預留時間的費用。它們是預付的，需要預先規劃容量需求，且不支援自動擴展，但會共置在 EC2 UltraClusters 中以進行低延遲聯網。它們只會在保留期間收費。若要進一步了解，請參閱尋找和購買容量區塊，或使用建立具有 ML 容量區塊的受管節點群組中的指示來設定具有容量區塊的受管節點群組。

透過 AWS 管理主控台預留容量，並將節點設定為使用 ML 容量區塊。根據工作負載排程規劃保留，並在預備叢集中進行測試。如需詳細資訊，請參閱容量區塊文件。

考慮 G Amazon EC2 執行個體的隨需、Amazon EC2 Spot 或隨需容量保留 (ODCRs)

對於 G Amazon EC2 執行個體，請考慮與隨需、Amazon EC2 Spot 執行個體和隨需容量保留不同的購買選項。ODCRs可讓您在特定的可用區域中保留 EC2 執行個體容量，即使在高需求期間也能確保可

用性。與僅適用於 P 和 Trainium 執行個體的 ML 容量區塊不同，ODCRs 可用於更廣泛的執行個體類型，包括 G 執行個體，使其適用於需要不同 GPU 功能的工作負載，例如推論或圖形。使用 Amazon EC2 Spot 執行個體時，能夠跨不同的執行個體類型、大小和可用區域進行多樣化，是能夠長時間停留在 Spot 上的關鍵。

ODCRs沒有長期承諾，但您要支付預留容量的隨需費率，無論是已使用或閒置。您可以建立 ODCRs 以供立即使用，或排定未來日期，提供容量規劃的彈性。在 Amazon EKS 中，Karpenter 和受管節點群組等節點類型支援 ODCRs。若要排定 Karpenter 中的 ODCRs優先順序，請將 NodeClass 設定為使用 `capacityReservationSelectorTerms` 欄位。請參閱 Karpenter NodePools 文件。如需建立 ODCRs 的詳細資訊，包括 CLI 命令，請參閱隨需容量保留入門。

考慮其他加速執行個體類型和大小

選取適當的加速執行個體和大小對於在 Amazon EKS 上最佳化 ML 工作負載的效能和成本至關重要。例如，不同的 GPU 執行個體系列具有不同的效能和功能，例如 GPU 記憶體。為了協助您選擇價格效能最佳的選項，請檢閱加速運算下 EC2 執行個體類型頁面中的可用 GPU 執行個體。評估多個執行個體類型和大小，以找出最適合您特定工作負載需求的執行個體類型和大小。考慮 GPUs 數量、記憶體和網路效能等因素。透過仔細選擇正確的 GPU 執行個體類型和大小，您可以在 EKS 叢集中實現更好的資源使用率和成本效益。

如果您在 EKS 節點中使用 GPU 執行個體，則預設在`kube-system`命名空間中會有 `nvidia-device-plugin-daemonset` Pod。若要快速了解您是否在執行個體中充分利用 GPU （如下所示），您可以使用 nvidia-smi：

```
kubectl exec nvidia-device-plugin-daemonset-xxxxx \
  -n kube-system -- nvidia-smi \
  --query-
gpu=index,power.draw,power.limit,temperature.gpu,utilization.gpu,utilization.memory,memory.free
 \
  --format=csv -l 5
```

- 如果 `utilization.memory` （接近 100%），則您的程式碼可能已繫結記憶體。這表示 GPU （記憶體）已充分利用，但可能表示應調查進一步的效能最佳化。
- 如果 `utilization.gpu` 接近 100%，這不一定表示 GPU 已充分利用。要查看的較佳指標是 `power.draw`與 的比率`power.limit`。如果此比率為 100% 或以上，則您的程式碼會充分利用 GPU 的運算容量 （運算容量）。
- `-l 5` 旗標指出 每 5 秒輸出一次指標。如果是單一 GPU 執行個體類型，則不需要索引查詢旗標。

若要進一步了解，請參閱 AWS 文件中的 GPU 執行個體。

使用時間分割、MIG 和分量 GPU 配置來最佳化 GPU 資源配置

Kubernetes 中的靜態資源限制（例如 CPU、記憶體、GPU 計數）可能會導致過度佈建或利用率不足，尤其是推論等動態 AI/ML 工作負載。選取正確的 GPU 非常重要。對於低容量或尖峰工作負載，時間分割允許多個工作負載透過共用其運算資源來共用單一 GPU，進而提高效率並減少浪費。GPU 共用可以透過不同的選項達成：

- 利用節點選擇器/節點親和性來影響排程：確保佈建的節點和 Pod 已排程在工作負載的適當 GPUs 上（例如 `karpenter.k8s.aws/instance-gpu-name: "a100"`）
- 時間分割：排程工作負載以隨時間共用 GPU 的運算資源，允許並行執行而不進行實體分割。這非常適合具有可變運算需求的工作負載，但可能缺少記憶體隔離。
- 多執行個體 GPU (MIG)：MIG 允許單一 NVIDIA GPU 分割成多個隔離的執行個體，並支援 NVIDIA Ampere（例如 A100 GPU)、NVIDIA Hopper（例如 H100 GPU) 和 NVIDIA Blackwell（例如 Blackwell GPUs) GPUs。每個 MIG 執行個體都會接收專用運算和記憶體資源，在需要資源保證的多租用戶環境或工作負載中啟用資源共用，這可讓您最佳化 GPU 資源使用率，包括透過時間分割提供不同批次大小的多個模型。
- 分數 GPU 分配：使用軟體型排程，將部分 GPU 運算或記憶體分配給工作負載，為動態工作負載提供彈性。NVIDIA KAI 排程器是 Run：ai 平台的一部分，透過允許 Pod 請求部分 GPU 資源來啟用此功能。

若要在 EKS 中啟用這些功能，您可以部署 NVIDIA 裝置外掛程式，將 GPUs 公開為可排程的資源，並支援時間分割和 MIG。若要進一步了解，請參閱 Kubernetes 中的時間分割 GPUs，以及在 Amazon EKS 上使用 NVIDIA 時間分割和加速 EC2 執行個體的 GPU 共用。

範例

例如，若要使用 NVIDIA 裝置外掛程式啟用時間分割：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-device-plugin-config
  namespace: kube-system
data:
  config.yaml: |
    version: v1
    sharing:
```

```
      timeSlicing:
        resources:
        - name: nvidia.com/gpu
          replicas: 4  # Allow 4 pods to share each GPU
```

範例

例如，若要使用 KAI 排程器配置分數 GPU，請將其與 NVIDIA GPU Operator 一起部署，並在 Pod 規格中指定分數 GPU 資源：

```
apiVersion: v1
kind: Pod
metadata:
  name: fractional-gpu-pod-example
  annotations:
    gpu-fraction: "0.5"  # Annotation for 50% GPU
  labels:
    runai/queue: "default"  # Required queue assignment
spec:
  containers:
  - name: ml-workload
    image: nvcr.io/nvidia/pytorch:25.04-py3
    resources:
      limits:
        nvidia.com/gpu: 1
  nodeSelector:
    nvidia.com/gpu: "true"
  schedulerName: kai-scheduler
```

節點彈性和訓練任務管理

使用自動復原實作節點運作狀態檢查

對於需要頻繁節點間通訊的 Amazon EKS 分散式訓練任務，例如跨多個節點的多 GPU 模型訓練，GPU 或 EFA 失敗等硬體問題可能會導致訓練任務中斷。這些中斷可能會導致訓練進度的損失和成本增加，尤其是依賴穩定硬體的長期執行 AI/ML 工作負載。

為了協助增加硬體故障的彈性，例如執行 GPU 工作負載的 EKS 叢集中的 GPU 失敗，我們建議您利用 EKS 節點監控代理程式搭配自動修復或 Amazon SageMaker HyperPod。雖然具有自動修復功能的 EKS 節點監控代理程式提供節點運作狀態監控和使用標準 Kubernetes 機制的自動修復等功能，但 SageMaker HyperPod 提供目標彈性和其他專為大規模 ML 訓練而設計的功能，例如深度運作狀態檢查和自動任務恢復。

- EKS Node Monitoring Agent with Node Auto Repair 透過讀取日誌和套用 NodeConditions 來持續監控節點運作狀態，包括加速硬體的特定標準條件，例如 Ready和 條件，以識別 GPU 或網路故障等問題。當節點視為運作狀態不佳時，Node Auto Repair 會將其封鎖，並將其取代為新的節點。重新排程 Pod 和重新啟動任務取決於標準 Kubernetes 機制和任務的重新啟動政策。
- SageMaker HyperPod 深度運作狀態檢查和運作狀態監控代理程式會持續監控 GPU 和 Trainium 型執行個體的運作狀態。它專為 AI/ML 工作負載量身打造，使用標籤 （ 例如 node-health-status) 來管理節點運作狀態。當節點視為運作狀態不佳時，HyperPod 會觸發自動取代故障的硬體，例如 GPUs。它預設會透過 EFA 的基本運作狀態檢查來偵測 EFA 的聯網相關故障，並支援針對中斷的訓練任務自動恢復，允許任務從最後一個檢查點繼續，將大規模 ML 任務的中斷降至最低。

對於具有自動修復的 EKS 節點監控代理程式和使用 EFA 的 SageMaker HyperPod 叢集，若要監控 EFA 特定指標，例如遠端直接記憶體存取 (RDMA) 錯誤和封包捨棄，請確定已安裝 AWS EFA 驅動程式。此外，我們建議部署 CloudWatch 可觀測性附加元件或使用 DCGM Exporter 等工具搭配 Prometheus 和 Grafana 來監控 EFA、GPU，以及針對 SageMaker HyperPod，與其功能相關的特定指標。

針對中斷敏感工作負載停用 Karpenter 合併

對於對處理、大規模 AI/ML 預測任務或訓練等中斷敏感的工作負載，我們建議調校 Karpenter 整合政策，以防止任務執行期間中斷。Karpenter 的合併功能會自動最佳化叢集成本，方法是終止未充分利用的節點，或以價格較低的替代節點取代它們。不過，即使工作負載完全利用 GPU，如果 Karpenter 識別出符合 Pod 需求的價格較低的適當執行個體類型，則可能會合併節點，導致任務中斷。

`WhenEmptyOrUnderutilized` 合併政策可能會提早終止節點，導致執行時間延長。例如，由於 Pod 重新排程、資料重新載入，中斷可能會延遲任務恢復，這對於長時間執行的批次推論任務可能非常昂貴。若要緩解這種情況，您可以將 `consolidationPolicy` 設定為 ，`WhenEmpty`並設定`consolidateAfter`持續時間，例如 1 小時，以在工作負載尖峰期間保留節點。例如：

```
disruption:
  consolidationPolicy: WhenEmpty
  consolidateAfter: 60m
```

此方法可改善尖峰批次推論工作負載和其他干擾敏感任務的 Pod 啟動延遲，例如即時線上推論資料處理或模型訓練，其中中斷成本高於運算成本節省。Karpenter NodePool 中斷預算是管理 Karpenter 中斷的另一項功能。透過預算，您可以確保在所選的 NodePool 中，不超過特定數量的節點節點會在某個時間點中斷。您也可以使用中斷預算來防止所有節點在特定時間 （ 例如尖峰時間 ） 中斷。若要進一步了解，請參閱 Karpenter 合併文件。

使用 ttlSecondsAfterFinished 自動清除 Kubernetes 任務

我們建議`ttlSecondsAfterFinished`在 Amazon EKS 中設定 Kubernetes 任務，以自動刪除已完成的任務物件。內嵌任務物件會耗用叢集資源，例如 API 伺服器記憶體，並透過雜亂儀表板（例如 Grafana、Amazon CloudWatch) 來複雜化監控。例如，將 TTL 設定為 1 小時可確保任務會在完成後不久移除，讓您的叢集保持整齊。如需詳細資訊，請參閱完成任務的自動清除。

設定較高優先順序任務/工作負載的低優先順序任務先佔

對於 Amazon EKS 上的混合優先順序 AI/ML 工作負載，您可以設定低優先順序任務先佔，以確保較高優先順序的任務（例如即時推論）可快速接收資源。如果沒有先佔，低優先順序工作負載，例如批次程序（例如批次推論、資料處理）、非批次服務（例如背景任務、cron 任務）或 CPU/記憶體密集型任務（例如 Web 服務），可以透過佔用節點來延遲關鍵 Pod。先佔允許 Kubernetes 在高優先順序 Pod 需要資源時移出低優先順序 Pod，確保在具有 GPUs、CPUs 或記憶體的節點上進行有效率的資源配置。建議使用 Kubernetes `PriorityClass`來指派優先順序`PodDisruptionBudget`並控制移出行為。

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: low-priority
value: 100
---
spec:
  priorityClassName: low-priority
```

如需詳細資訊，請參閱 Kubernetes Priority and Preemption Documentation。

應用程式擴展和效能

使用 Karpenter 或靜態節點的 ML 工作負載量身訂做運算容量

為了確保 Amazon EKS 上機器學習 (ML) 工作流程的成本效益和回應性運算容量，我們建議您根據工作負載的特性和成本承諾量身打造節點佈建策略。以下是兩種考慮方法：使用 Karpenter 和靜態節點群組即時 just-in-time 擴展預留容量。

- Just-in-time資料平面縮放器，例如 Karpenter：對於具有可變運算需求的動態 ML 工作流程（例如，GPU 型推論，後面接著 CPU 型繪圖），我們建議使用just-in-time資料平面縮放器，例如 Karpenter。

- 將靜態節點群組用於可預測的工作負載：對於可預測、穩定狀態的 ML 工作負載，或使用預留執行個體時，EKS 受管節點群組可協助確保預留容量已完全佈建和使用，從而最大限度地節省成本。此方法非常適合透過 RIs 或 ODCRs 遞交的特定執行個體類型。

範例

這是各種 Karpenter NodePool 的範例，可啟動執行個體產生大於三個的 g Amazon EC2 執行個體。

```
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: gpu-inference
spec:
  template:
    spec:
      nodeClassRef:
        group: karpenter.k8s.aws
        kind: EC2NodeClass
        name: default
      requirements:
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["on-demand"]
        - key: karpenter.k8s.aws/instance-category
          operator: In
          values: ["g"]
        - key: karpenter.k8s.aws/instance-generation
          operator: Gt
          values: ["3"]
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"]
      taints:
        - key: nvidia.com/gpu
          effect: NoSchedule
  limits:
    cpu: "1000"
    memory: "4000Gi"
    nvidia.com/gpu: "10"  *# Limit the total number of GPUs to 10 for the NodePool*
  disruption:
    consolidationPolicy: WhenEmpty
    consolidateAfter: 60m
```

```
    expireAfter: 720h
```

範例

使用靜態節點群組訓練工作負載的範例：

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig
metadata:
  name: ml-cluster
  region: us-west-2
managedNodeGroups:
  - name: gpu-node-group
    instanceType: p4d.24xlarge
    minSize: 2
    maxSize: 2
    desiredCapacity: 2
    taints:
      - key: nvidia.com/gpu
        effect: NoSchedule
```

使用污點和容錯，防止在加速執行個體上排程非加速工作負載

在 GPU 資源上排程非加速工作負載不具運算效率，我們建議您使用污點和容錯，以確保非加速工作負載 Pod 不會排程在不適當的節點上。如需詳細資訊，請參閱 Kubernetes 文件。

根據模型效能擴展

對於推論工作負載，我們建議您使用 Kubernetes Event-Driven Autoscaling (KEDA) 根據模型效能指標進行擴展，例如推論請求或字符輸送量，並具有適當的冷卻時間。靜態擴展政策可能會過度佈建或佈建不足的資源，影響成本和延遲。如需進一步了解，請參閱 KEDA 文件。

進階 GPU 管理的動態資源配置

動態資源配置 (DRA) 代表 Kubernetes GPU 資源管理的基本進展。DRA 超越了傳統的裝置外掛程式限制，以啟用複雜的 GPU 共用、拓撲感知和跨節點資源協調。適用於 Amazon EKS 1.33 版的 DRA 透過提供下列項目來解決 AI/ML 工作負載中的重大挑戰：

- 精細的 GPU 配置
- 進階共用機制，例如多程序服務 (MPS) 和多執行個體 GPU (MIG)

- 支援新一代硬體架構，包括 NVIDIA GB200 UltraServers

傳統 GPU 配置會將 GPUs 視為不透明的整數資源，造成顯著的使用不足（通常在生產叢集中為 30-40%)。這是因為即使只需要部分資源，工作負載也會取得整個 GPUs 的專屬存取權。DRA 透過引入結構化的宣告式配置來轉換此模型，讓 Kubernetes 排程器完全了解硬體特性和工作負載需求。這可實現智慧型配置決策和有效率的資源共用。

使用 DRA 而非 NVIDIA 裝置外掛程式的優點

NVIDIA 裝置外掛程式（從版本 開始`0.12.0`) 支援 GPU 共用機制，包括時間分割、MPS 和 MIG。不過，該 DRA 地址存在架構限制。

NVIDIA 裝置外掛程式限制

- 靜態組態：GPU 共用組態（時間分割複本和 MPS 設定）需要透過 進行整個叢集的預先組態`ConfigMaps`。這使得為不同的工作負載提供不同的共用策略變得困難。
- 有限精細選擇：雖然裝置外掛程式透過節點標籤公開 GPU 特性，但工作負載無法在排程決策中動態請求特定 GPU 組態（記憶體大小和運算功能）。
- 無跨節點資源協調：無法管理跨多個節點的分散式 GPU 資源，或表達複雜的拓撲需求，例如 NVIDIA GB200 等系統的 NVLink 網域。
- 排程器限制：Kubernetes 排程器會將 GPU 資源視為不透明整數，限制其做出拓撲感知決策或處理複雜資源相依性的能力。
- 組態複雜性：設定不同的共用策略需要多個仔細`ConfigMaps`的節點標記，從而產生操作複雜性。

搭配 DRA 的解決方案

- 動態資源選擇：DRA 允許工作負載在透過 的請求時間指定詳細需求 (GPU 記憶體、驅動程式版本和特定屬性）`resourceclaims`。這可讓資源比對更具彈性。
- 拓撲感知：透過結構化參數和裝置選擇器，DRA 可處理複雜的需求，例如跨節點 GPU 通訊和記憶體一致性互連。
- 跨節點資源管理：`computeDomains`啟用跨多個節點的分散式 GPU 資源協調，對於具有 IMEX 通道的 GB200 等系統至關重要。
- 工作負載特定的組態：每個 `ResourceClaim` 指定不同的共用策略和組態，允許每個工作負載的精細控制，而不是整個叢集的設定。
- 增強型排程器整合：DRA 提供排程器詳細的裝置資訊，並根據硬體拓撲和資源特性進行更明智的置放決策。

重要：DRA 不會完全取代 NVIDIA 裝置外掛程式。NVIDIA DRA 驅動程式可與裝置外掛程式搭配使用，以提供增強功能。裝置外掛程式會繼續處理基本的 GPU 探索和管理，而 DRA 新增進階配置和排程功能。

DRA 支援的執行個體及其功能

DRA 支援因 Amazon EC2 執行個體系列和 GPU 架構而異，如下表所示。

執行個體系列	GPU 類型	時間分割	MIG 支援	MPS 支援	IMEX 支援	使用案例
G5	NVIDIA A10G	是	否	是	否	推論和圖形工作負載
G6	NVIDIA L4	是	否	是	否	AI 推論和影片處理
G6e	NVIDIA L40S	是	否	是	否	訓練、推論和圖形
P4d/ P4de	NVIDIA A100	是	是	是	否	大規模訓練和 HPC
P5	NVIDIA H100	是	是	是	否	基礎模型訓練
P6	NVIDIA B200	是	是	是	否	十億或兆參數模型、分散式訓練和推論
P6e	NVIDIA GB200	是	是	是	是	十億或兆參數模型、分散式訓練和推論

以下是 資料表中每個功能的描述：

- 時間分割：允許多個工作負載隨時間共用 GPU 運算資源。
- 多執行個體 GPU (MIG)：建立隔離 GPU 執行個體的硬體層級分割。
- 多程序服務 (MPS)：在單一 GPU 上啟用多個 CUDA 程序的並行執行。
- 節點間記憶體交換 (IMEX)：GB200 UltraServers 跨節點的記憶體一致性通訊。

其他資源

如需 Kubernetes DRA 和 NVIDIA DRA 驅動程式的詳細資訊，請參閱 GitHub 上的下列資源：

- Kubernetes dynamic-resource-allocation
- DRA 的 Kubernetes 增強提案
- 適用於 GPUs 的 NVIDIA DRA 驅動程式
- NVIDIA DRA 範例和快速入門

設定動態資源配置以進行進階 GPU 管理

下列主題說明如何設定動態資源配置 (DRA) 以進行進階 GPU 管理。

先決條件

在 Amazon EKS 上實作 DRA 之前，請確定您的環境符合下列要求。

叢集組態

- 執行版本 `1.33`或更新版本的 Amazon EKS 叢集
- Amazon EKS 受管節點群組 (DRA 目前僅支援具有 AL2023 和 Bottlerocket NVIDIA 最佳化 AMIs受管節點群組，不支援 Karpenter)
- 具有適當執行個體類型的 NVIDIA GPU 工作者節點

必要元件

- NVIDIA 裝置外掛程式版本 `0.17.1` 或更新版本
- NVIDIA DRA 驅動程式版本 `25.3.0` 或更新版本

步驟 1：使用 eksctl 使用已啟用 DRA 的節點群組建立叢集

1. 建立名為 的叢集組態檔案`dra-eks-cluster.yaml`：

```
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dra-eks-cluster
```

```
  region: us-west-2
  version: '1.33'

managedNodeGroups:
- name: gpu-dra-nodes
  amiFamily: AmazonLinux2023
  instanceType: g6.12xlarge
  desiredCapacity: 2
  minSize: 1
  maxSize: 3

  labels:
    node-type: "gpu-dra"
    nvidia.com/gpu.present: "true"

  taints:
  - key: nvidia.com/gpu
    value: "true"
    effect: NoSchedule
```

2. 建立叢集：

```
eksctl create cluster -f dra-eks-cluster.yaml
```

步驟 2：部署 NVIDIA 裝置外掛程式

部署 NVIDIA 裝置外掛程式以啟用基本 GPU 探索：

1. 新增 NVIDIA 裝置外掛程式 Helm 儲存庫：

```
helm repo add nvidia https://nvidia.github.io/k8s-device-plugin
helm repo update
```

2. 為裝置外掛程式建立自訂值：

```
cat <<EOF > nvidia-device-plugin-values.yaml
gfd:
  enabled: true
nfd:
  enabled: true
tolerations:
  - key: nvidia.com/gpu
```

```
    operator: Exists
    effect: NoSchedule
EOF
```

3. 安裝 NVIDIA 裝置外掛程式：

```
helm install nvidia-device-plugin nvidia/nvidia-device-plugin \
 --namespace nvidia-device-plugin \
 --create-namespace \
 --version 0.17.1 \
 --values nvidia-device-plugin-values.yaml
```

步驟 3：部署 NVIDIA DRA 驅動程式 Helm Chart

1. 建立 DRA 驅動程式`dra-driver-values.yaml`的值檔案：

```
---
nvidiaDriverRoot: /

gpuResourcesEnabledOverride: true

resources:
  gpus:
    enabled: true
  computeDomains:
    enabled: true  # Enable for GB200 IMEX support

controller:
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule

kubeletPlugin:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: "nvidia.com/gpu.present"
            operator: In
            values: ["true"]
```

```
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

2. 新增 NVIDIA NGC Helm 儲存庫：

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia
helm repo update
```

3. 安裝 NVIDIA DRA 驅動程式：

```
helm install nvidia-dra-driver nvidia/nvidia-dra-driver-gpu \
 --version="25.3.0-rc.2" \
 --namespace nvidia-dra-driver \
 --create-namespace \
 --values dra-driver-values.yaml
```

步驟 4：驗證 DRA 安裝

1. 確認 DRA API 資源可用：

```
kubectl api-resources | grep resource.k8s.io/v1beta1
```

以下是預期的輸出：

```
deviceclasses resource.k8s.io/v1beta1 false DeviceClass
resourceclaims resource.k8s.io/v1beta1 true ResourceClaim
resourceclaimtemplates resource.k8s.io/v1beta1 true ResourceClaimTemplate
resourceslices resource.k8s.io/v1beta1 false ResourceSlice
```

2. 檢查可用的裝置類別：

```
kubectl get deviceclasses
```

以下是預期輸出的範例：

```
NAME                                        AGE
compute-domain-daemon.nvidia.com            4h39m
compute-domain-default-channel.nvidia.com   4h39m
```

```
gpu.nvidia.com                                  4h39m
mig.nvidia.com                                  4h39m
```

當新建立的 G6 GPU 執行個體加入已啟用 DRA 的 Amazon EKS 叢集時，會發生下列動作：

- NVIDIA DRA 驅動程式會自動探索 A10G GPU，並在該節點`resourceslices`上建立兩個 GPU。
- `gpu.nvidia.com` 配量會將實體 A10G GPU 裝置與其規格（記憶體、運算功能等）註冊。
- 由於 A10G 不支援 MIG 分割，`compute-domain.nvidia.com`配量會建立代表 GPU 整個運算內容的單一運算網域。
- 這些`resourceslices`接著會發佈至 Kubernetes API 伺服器，讓 GPU 資源可透過 進行排程`resourceclaims`。

  DRA 排程器現在可以智慧地將此 GPU 配置給透過 請求 GPU 資源的 Pod`resourceclaimtemplates`，提供比傳統裝置外掛程式方法更靈活的資源管理。這會自動發生，無需手動介入。一旦 DRA 驅動程式完成資源探索和註冊程序，節點即可直接用於 GPU 工作負載。

  當您執行下列命令時：

```
kubectl get resourceslices
```

  以下是預期輸出的範例：

```
NAME                                                                NODE
            DRIVER                          POOL                                    AGE
ip-100-64-129-47.ec2.internal-compute-domain.nvidia.com-rwsts
 ip-100-64-129-47.ec2.internal    compute-domain.nvidia.com
 ip-100-64-129-47.ec2.internal    35m
ip-100-64-129-47.ec2.internal-gpu.nvidia.com-6kndg
 ip-100-64-129-47.ec2.internal    gpu.nvidia.com
 ip-100-64-129-47.ec2.internal    35m
```

繼續進行the section called “使用動態資源配置排程簡單的 GPU 工作負載”。

使用動態資源配置排程簡單的 GPU 工作負載

若要使用動態資源配置 (DRA) 排程簡單的 GPU 工作負載，請執行下列步驟。在繼續之前，請確定您已遵循 the section called “設定動態資源配置以進行進階 GPU 管理”。

1. 使用名為 ResourceClaimTemplate 的檔案建立 GPU 配置的基本basic-gpu-claim-template.yaml：

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: gpu-test1

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  namespace: gpu-test1
  name: single-gpu
spec:
  spec:
    devices:
      requests:
      - name: gpu
        deviceClassName: gpu.nvidia.com
```

2. 套用範本：

```
kubectl apply -f basic-gpu-claim-template.yaml
```

3. 驗證狀態：

```
kubectl get resourceclaimtemplates -n gpu-test1
```

下列為範例輸出：

```
NAME         AGE
single-gpu   9m16s
```

4. 建立使用 ResourceClaimTemplate 檔案名為 的 Podbasic-gpu-pod.yaml：

```
---
apiVersion: v1
kind: Pod
metadata:
  namespace: gpu-test1
```

```
  name: gpu-pod
  labels:
    app: pod
spec:
  containers:
  - name: ctr0
    image: ubuntu:22.04
    command: ["bash", "-c"]
    args: ["nvidia-smi -L; trap 'exit 0' TERM; sleep 9999 & wait"]
    resources:
      claims:
      - name: gpu0
  resourceClaims:
  - name: gpu0
    resourceClaimTemplateName: single-gpu
  nodeSelector:
    NodeGroupType: gpu-dra
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: "nvidia.com/gpu"
    operator: "Exists"
    effect: "NoSchedule"
```

5. 套用和監控 Pod：

```
kubectl apply -f basic-gpu-pod.yaml
```

6. 檢查 Pod 狀態：

```
kubectl get pod -n gpu-test1
```

以下是預期的輸出範例：

```
NAME      READY   STATUS    RESTARTS   AGE
gpu-pod   1/1     Running   0          13m
```

7. 檢查ResourceClaim狀態：

```
kubectl get resourceclaims -n gpu-test1
```

以下是預期的輸出範例：

```
NAME                  STATE                AGE
gpu-pod-gpu0-l76cg    allocated,reserved   9m6s
```

8. 檢視 Pod 日誌以查看 GPU 資訊：

```
kubectl logs gpu-pod -n gpu-test1
```

以下是預期的輸出範例：

```
GPU 0: NVIDIA L4 (UUID: GPU-da7c24d7-c7e3-ed3b-418c-bcecc32af7c5)
```

繼續使用 DRA 繼續取得更進階the section called “具有動態資源配置的 GPU 最佳化技術”的 GPU 最佳化技術。

具有動態資源配置的 GPU 最佳化技術

現代 GPU 工作負載需要複雜的資源管理，才能達到最佳的使用率和成本效益。DRA 啟用數種進階最佳化技術，可處理不同的使用案例和硬體功能：

- 時間分割可讓多個工作負載隨時間共用 GPU 運算資源，因此非常適合偶爾使用 GPU 的推論工作負載。如需範例，請參閱 the section called “使用時間分割最佳化 GPU 工作負載”。
- 多程序服務 (MPS) 可在單一 GPU 上並行執行多個 CUDA 程序，與時間分割相比具有更好的隔離能力。如需範例，請參閱 the section called “使用 MPS 最佳化 GPU 工作負載”。
- 多執行個體 GPU (MIG) 提供硬體層級分割，建立具有專用運算和記憶體資源的隔離 GPU 執行個體。如需範例，請參閱 the section called “使用多執行個體 GPU 最佳化 GPU 工作負載”。
- Internode Memory Exchange (IMEX) 可讓跨節點的記憶體一致性通訊，以便在 NVIDIA GB200 系統上進行分散式訓練。如需範例，請參閱 the section called “使用 GB200 P6e 執行個體透過 IMEX 最佳化 GPU 工作負載”。

這些技術可以大幅改善資源使用率。Organizations 報告 GPU 使用率從傳統配置的 30-40% 增加到最佳化共用策略的 80-90%。技術選擇取決於工作負載特性、隔離要求和硬體功能。

使用時間分割最佳化 GPU 工作負載

時間分割可讓多個工作負載透過排程在相同的實體 GPU 上循序執行，來共用 GPU 運算資源。它非常適合使用零星 GPU 的推論工作負載。

執行下列步驟。

1. 使用名為 的檔案定義時間分割ResourceClaimTemplate的 timeslicing-claim-template.yaml：

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: timeslicing-gpu

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: timeslicing-gpu-template
  namespace: timeslicing-gpu
spec:
  spec:
    devices:
      requests:
      - name: shared-gpu
        deviceClassName: gpu.nvidia.com
      config:
      - requests: ["shared-gpu"]
        opaque:
          driver: gpu.nvidia.com
          parameters:
            apiVersion: resource.nvidia.com/v1beta1
            kind: GpuConfig
            sharing:
              strategy: TimeSlicing
```

2. 使用名為 的檔案，使用時間分割來定義 Podtimeslicing-pod.yaml：

```
---
# Pod 1 - Inference workload
apiVersion: v1
kind: Pod
metadata:
  name: inference-pod-1
  namespace: timeslicing-gpu
  labels:
```

```
    app: gpu-inference
spec:
  restartPolicy: Never
  containers:
  - name: inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import time
      import os
      print(f"=== POD 1 STARTING ===")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")
      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
 {torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")
          # Simulate inference workload
          for i in range(20):
              x = torch.randn(1000, 1000).cuda()
              y = torch.mm(x, x.t())
              print(f"Pod 1 - Iteration {i+1} completed at {time.strftime('%H:%M:
%S')}")
              time.sleep(60)
      else:
          print("No GPU available!")
          time.sleep(5)
    resources:
      claims:
      - name: shared-gpu-claim
  resourceClaims:
  - name: shared-gpu-claim
    resourceClaimTemplateName: timeslicing-gpu-template
  nodeSelector:
    NodeGroupType: "gpu-dra"
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

```
---
# Pod 2 - Training workload
apiVersion: v1
kind: Pod
metadata:
  name: training-pod-2
  namespace: timeslicing-gpu
  labels:
    app: gpu-training
spec:
  restartPolicy: Never
  containers:
  - name: training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import time
      import os
      print(f"=== POD 2 STARTING ===")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")
      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
 {torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")
          # Simulate training workload with heavier compute
          for i in range(15):
              x = torch.randn(2000, 2000).cuda()
              y = torch.mm(x, x.t())
              loss = torch.sum(y)
              print(f"Pod 2 - Training step {i+1}, Loss: {loss.item():.2f} at
 {time.strftime('%H:%M:%S')}")
              time.sleep(5)
      else:
          print("No GPU available!")
          time.sleep(60)
    resources:
      claims:
      - name: shared-gpu-claim-2
  resourceClaims:
```

```
  - name: shared-gpu-claim-2
    resourceClaimTemplateName: timeslicing-gpu-template
  nodeSelector:
    NodeGroupType: "gpu-dra"
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

3. 套用範本和 Pod：

```
kubectl apply -f timeslicing-claim-template.yaml
kubectl apply -f timeslicing-pod.yaml
```

4. 監控資源宣告：

```
kubectl get resourceclaims -n timeslicing-gpu -w
```

下列為範例輸出：

```
NAME                                        STATE                AGE
inference-pod-1-shared-gpu-claim-9p97x      allocated,reserved   21s
training-pod-2-shared-gpu-claim-2-qghnb     pending              21s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
training-pod-2-shared-gpu-claim-2-qghnb     pending              105s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
training-pod-2-shared-gpu-claim-2-qghnb     allocated,reserved   105s
inference-pod-1-shared-gpu-claim-9p97x      pending              105s
```

第一個 Pod (`inference-pod-1`)

- 狀態： `allocated,reserved`
- 意義：DRA 找到可用的 GPU，並預留給此 Pod
- Pod 狀態：立即開始執行

第二個 Pod (`training-pod-2`)

- 狀態： `pending`

- 意義：等待 DRA 在相同 GPU 上設定時間分割
- Pod 狀態：等待排程
- 狀態將從 移至 pending allocated,reserved至 running

使用 MPS 最佳化 GPU 工作負載

多程序服務 (MPS) 可在單一 GPU 上並行執行多個 CUDA 內容，其隔離程度優於時間分割。

執行下列步驟。

1. 使用名為 的檔案為 ResourceClaimTemplate MPS 定義 mps-claim-template.yaml：

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: mps-gpu

---
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mps-gpu-template
  namespace: mps-gpu
spec:
  spec:
    devices:
      requests:
      - name: shared-gpu
        deviceClassName: gpu.nvidia.com
      config:
      - requests: ["shared-gpu"]
        opaque:
          driver: gpu.nvidia.com
          parameters:
            apiVersion: resource.nvidia.com/v1beta1
            kind: GpuConfig
            sharing:
              strategy: MPS
```

2. 使用 MPS 搭配名為 的檔案來定義 Podmps-pod.yaml：

```
---
# Single Pod with Multiple Containers sharing GPU via MPS
apiVersion: v1
kind: Pod
metadata:
  name: mps-multi-container-pod
  namespace: mps-gpu
  labels:
    app: mps-demo
spec:
  restartPolicy: Never
  containers:
  # Container 1 - Inference workload
  - name: inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "-c"]
    args:
    - |
      import torch
      import torch.nn as nn
      import time
      import os

      print(f"=== INFERENCE CONTAINER STARTING ===")
      print(f"Process ID: {os.getpid()}")
      print(f"GPU available: {torch.cuda.is_available()}")
      print(f"GPU count: {torch.cuda.device_count()}")

      if torch.cuda.is_available():
          device = torch.cuda.current_device()
          print(f"Current GPU: {torch.cuda.get_device_name(device)}")
          print(f"GPU Memory:
 {torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")

          # Create inference model
          model = nn.Sequential(
              nn.Linear(1000, 500),
              nn.ReLU(),
              nn.Linear(500, 100)
          ).cuda()

          # Run inference
          for i in range(1, 999999):
```

```
            with torch.no_grad():
                x = torch.randn(128, 1000).cuda()
                output = model(x)
                result = torch.sum(output)
                print(f"Inference Container PID {os.getpid()}: Batch {i}, Result:
{result.item():.2f} at {time.strftime('%H:%M:%S')}")
            time.sleep(2)
    else:
        print("No GPU available!")
        time.sleep(60)
  resources:
    claims:
    - name: shared-gpu-claim
      request: shared-gpu

 # Container 2 - Training workload
 - name: training-container
   image: nvcr.io/nvidia/pytorch:25.04-py3
   command: ["python", "-c"]
   args:
   - |
     import torch
     import torch.nn as nn
     import time
     import os

     print(f"=== TRAINING CONTAINER STARTING ===")
     print(f"Process ID: {os.getpid()}")
     print(f"GPU available: {torch.cuda.is_available()}")
     print(f"GPU count: {torch.cuda.device_count()}")

     if torch.cuda.is_available():
         device = torch.cuda.current_device()
         print(f"Current GPU: {torch.cuda.get_device_name(device)}")
         print(f"GPU Memory:
{torch.cuda.get_device_properties(device).total_memory / 1024**3:.1f} GB")

         # Create training model
         model = nn.Sequential(
             nn.Linear(2000, 1000),
             nn.ReLU(),
             nn.Linear(1000, 500),
             nn.ReLU(),
             nn.Linear(500, 10)
```

```
            ).cuda()

            criterion = nn.MSELoss()
            optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

            # Run training
            for epoch in range(1, 999999):
                x = torch.randn(64, 2000).cuda()
                target = torch.randn(64, 10).cuda()

                optimizer.zero_grad()
                output = model(x)
                loss = criterion(output, target)
                loss.backward()
                optimizer.step()

                print(f"Training Container PID {os.getpid()}: Epoch {epoch}, Loss:
{loss.item():.4f} at {time.strftime('%H:%M:%S')}")
                time.sleep(3)
        else:
            print("No GPU available!")
            time.sleep(60)
      resources:
        claims:
        - name: shared-gpu-claim
          request: shared-gpu

    resourceClaims:
    - name: shared-gpu-claim
      resourceClaimTemplateName: mps-gpu-template

    nodeSelector:
      NodeGroupType: "gpu-dra"
      nvidia.com/gpu.present: "true"
    tolerations:
    - key: nvidia.com/gpu
      operator: Exists
      effect: NoSchedule
```

3. 套用範本並建立多個 MPS Pod：

```
kubectl apply -f mps-claim-template.yaml
kubectl apply -f mps-pod.yaml
```

4. 監控資源宣告：

```
kubectl get resourceclaims -n mps-gpu -w
```

下列為範例輸出：

```
NAME                                                STATE                AGE
mps-multi-container-pod-shared-gpu-claim-2p9kx   allocated,reserved   86s
```

此組態透過動態資源分配 (DRA) 使用 NVIDIA Multi-Process Service (MPS) 示範真正的 GPU 共用。與使用 GPU 依序輪換工作負載的時間分割不同，MPS 可讓兩個容器同時在相同的實體 GPU 上執行。主要洞見是 DRA MPS 共用需要單一 Pod 中的多個容器，而不是多個單獨的 Pod。部署時，DRA 驅動程式會將一個配置`ResourceClaim`給 Pod，並自動設定 MPS，以允許推論和訓練容器同時執行。

每個容器都有自己的隔離 GPU 記憶體空間和運算資源，搭配 MPS 協助程式來協調基礎硬體的存取。您可以執行下列動作來驗證是否正常運作：

- 檢查 `nvidia-smi`，這會將兩個容器顯示為共用相同 GPU 裝置的 M+C (`MPS + Compute`) 程序。
- 從兩個容器監控日誌，這會顯示可同時執行的交錯時間戳記。

這種方法允許互補工作負載有效地共用昂貴的 GPU 硬體，而不是讓單一程序未充分利用 GPU 使用率，從而最大限度地提高 GPU 使用率。

Container1： **`inference-container`**

```
root@mps-multi-container-pod:/workspace# nvidia-smi
Wed Jul 16 21:09:30 2025
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01             Driver Version: 570.158.01     CUDA Version: 12.9
   |
|-----------------------------------------+------------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr.
 ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute
 M. |
|                                         |                        |               MIG
 M. |
```

```
|=========================================+========================
+======================|
|   0  NVIDIA L4                      On  |   00000000:35:00.0 Off |
 0 |
| N/A   48C    P0             28W /   72W |     597MiB /  23034MiB |      0%   E.
 Process |
|                                         |                        |
 N/A |
+-----------------------------------------+------------------------
+----------------------+

+-----------------------------------------------------------------------------------------
+
| Processes:
   |
|  GPU   GI   CI              PID   Type   Process name                        GPU
 Memory |
|        ID   ID                                                               Usage
   |
|
=========================================================================================|
|    0   N/A  N/A               1    M+C   python
 246MiB |
+-----------------------------------------------------------------------------------------
+
```

Container2： **`training-container`**

```
root@mps-multi-container-pod:/workspace# nvidia-smi
Wed Jul 16 21:16:00 2025
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01             Driver Version: 570.158.01     CUDA Version: 12.9
   |
|-----------------------------------------+------------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr.
 ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute
 M. |
|                                         |                        |               MIG
 M. |
```

```
|=========================================+========================
+======================|
|   0  NVIDIA L4                      On  |   00000000:35:00.0 Off |
 0 |
| N/A   51C    P0             28W /   72W |     597MiB /  23034MiB |      0%   E.
 Process |
|                                         |                        |
 N/A |
+-----------------------------------------+------------------------
+----------------------+

+---------------------------------------------------------------------------------------
+
| Processes:
   |
|  GPU   GI   CI              PID   Type   Process name                        GPU
 Memory |
|        ID   ID                                                               Usage
   |
|
=======================================================================================|
|    0   N/A  N/A               1    M+C   python
 314MiB |
+---------------------------------------------------------------------------------------
+
```

使用多執行個體 GPU 最佳化 GPU 工作負載

多執行個體 GPU (MIG) 提供硬體層級分割，建立具有專用運算和記憶體資源的隔離 GPU 執行個體。

搭配各種設定檔使用動態 MIG 分割需要 NVIDIA GPU Operator。NVIDIA GPU Operator 使用 MIG Manager 建立 MIG 設定檔，並重新啟動 P4D、P4De、P5、P6 等 GPU 執行個體，以套用組態變更。GPU Operator 透過 MIG Manager 元件包含完整的 MIG 管理功能，可監控節點標籤變更，並自動套用適當的 MIG 組態。請求 MIG 設定檔變更時，運算子會正常關閉所有 GPU 用戶端、套用新的分割區幾何，並重新啟動受影響的服務。此程序需要重新啟動 GPU 執行個體的節點，以確保 GPU 狀態的轉換乾淨。這就是為什麼在 MIG Manager 組態`WITH0REB00T=true`中啟用 對成功的 MIG 部署至關重要。

您需要 NVIDIA DRA 驅動程式和 NVIDIA GPU Operator 才能在 Amazon EKS 中使用 MIG。除此之外，您不需要 NVIDIA 裝置外掛程式和 DCGM Exporter，因為這些是 NVIDIA GPU Operator 的一部分。由於 EKS NVIDIA AMIs 隨附預先安裝的 NVIDIA 驅動程式，因此我們停用 GPU Operator 部署驅動程式，以避免衝突並利用執行個體上已存在的最佳化驅動程式。NVIDIA DRA 驅動程式會處理 MIG

執行個體的動態資源配置，而 GPU Operator 會管理整個 GPU 生命週期。這包括 MIG 組態、裝置外掛程式功能、透過 DCGM 監控，以及節點功能探索。這種整合方法為企業 GPU 管理提供了完整的解決方案，具有硬體層級隔離和動態資源配置功能。

步驟 1：部署 NVIDIA GPU Operator

1. 新增 NVIDIA GPU Operator 儲存庫：

```
helm repo add nvidia https://nvidia.github.io/gpu-operator
helm repo update
```

2. 建立gpu-operator-values.yaml檔案：

```
driver:
  enabled: false

mig:
  strategy: mixed

migManager:
  enabled: true
  env:
    - name: WITH_REBOOT
      value: "true"
  config:
    create: true
    name: custom-mig-parted-configs
    default: "all-disabled"
    data:
      config.yaml: |-
        version: v1
        mig-configs:
          all-disabled:
            - devices: all
              mig-enabled: false

          # P4D profiles (A100 40GB)
          p4d-half-balanced:
            - devices: [0, 1, 2, 3]
              mig-enabled: true
              mig-devices:
                "1g.5gb": 2
                "2g.10gb": 1
```

```
                "3g.20gb": 1
            - devices: [4, 5, 6, 7]
              mig-enabled: false

          # P4DE profiles (A100 80GB)
          p4de-half-balanced:
            - devices: [0, 1, 2, 3]
              mig-enabled: true
              mig-devices:
                "1g.10gb": 2
                "2g.20gb": 1
                "3g.40gb": 1
            - devices: [4, 5, 6, 7]
              mig-enabled: false

devicePlugin:
  enabled: true
  config:
    name: ""
    create: false
    default: ""

toolkit:
  enabled: true

nfd:
  enabled: true

gfd:
  enabled: true

dcgmExporter:
  enabled: true
  serviceMonitor:
    enabled: true
    interval: 15s
    honorLabels: false
    additionalLabels:
      release: kube-prometheus-stack

nodeStatusExporter:
  enabled: false

operator:
```

```
  defaultRuntime: containerd
  runtimeClass: nvidia
  resources:
    limits:
      cpu: 500m
      memory: 350Mi
    requests:
      cpu: 200m
      memory: 100Mi

daemonsets:
  tolerations:
    - key: "nvidia.com/gpu"
      operator: "Exists"
      effect: "NoSchedule"
  nodeSelector:
    accelerator: nvidia
  priorityClassName: system-node-critical
```

3. 使用 `gpu-operator-values.yaml` 檔案安裝 GPU Operator：

```
helm install gpu-operator nvidia/gpu-operator \
  --namespace gpu-operator \
  --create-namespace \
  --version v25.3.1 \
  --values gpu-operator-values.yaml
```

此 Helm Chart 部署下列元件和多個 MIG 設定檔：

- 裝置外掛程式 (GPU 資源排程）
- DCGM Exporter (GPU 指標和監控）
- 節點功能探索 (NFD - 硬體標籤）
- GPU 功能探索 (GFD - GPU 特定標籤）
- MIG Manager （多執行個體 GPU 分割）
- Container Toolkit (GPU 容器執行時間）
- 運算子控制器（生命週期管理）

4. 驗證部署 Pod：

```
kubectl get pods -n gpu-operator
```

下列為範例輸出：

```
NAME                                                              READY   STATUS
 RESTARTS        AGE
gpu-feature-discovery-27rdq                                       1/1     Running
 0               3h31m
gpu-operator-555774698d-48brn                                     1/1     Running
 0               4h8m
nvidia-container-toolkit-daemonset-sxmh9                          1/1     Running
 1 (3h32m ago)   4h1m
nvidia-cuda-validator-qb77g                                       0/1     Completed
 0               3h31m
nvidia-dcgm-exporter-cvzd7                                        1/1     Running
 0               3h31m
nvidia-device-plugin-daemonset-5ljm5                              1/1     Running
 0               3h31m
nvidia-gpu-operator-node-feature-discovery-gc-67f66fc557-q5wkt    1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-master-5d8ffddcsl6s6   1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-worker-6t4w7           1/1     Running
 1 (3h32m ago)   4h1m
nvidia-gpu-operator-node-feature-discovery-worker-9w7g8           1/1     Running
 0               4h8m
nvidia-gpu-operator-node-feature-discovery-worker-k5fgs           1/1     Running
 0               4h8m
nvidia-mig-manager-zvf54                                          1/1     Running
 1 (3h32m ago)   3h35m
```

5. 使用 p4De 受管節點群組建立 Amazon EKS 叢集，以測試 MIG 範例：

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: dra-eks-cluster
  region: us-east-1
  version: '1.33'

managedNodeGroups:
# P4DE MIG Node Group with Capacity Block Reservation
- name: p4de-mig-nodes
```

```
amiFamily: AmazonLinux2023
instanceType: p4de.24xlarge

# Capacity settings
desiredCapacity: 0
minSize: 0
maxSize: 1

# Use specific subnet in us-east-1b for capacity reservation
subnets:
  - us-east-1b

# AL2023 NodeConfig for RAID0 local storage only
nodeadmConfig:
  apiVersion: node.eks.aws/v1alpha1
  kind: NodeConfig
  spec:
    instance:
      localStorage:
        strategy: RAID0

# Node labels for MIG configuration
labels:
  nvidia.com/gpu.present: "true"
  nvidia.com/gpu.product: "A100-SXM4-80GB"
  nvidia.com/mig.config: "p4de-half-balanced"
  node-type: "p4de"
  vpc.amazonaws.com/efa.present: "true"
  accelerator: "nvidia"

# Node taints
taints:
  - key: nvidia.com/gpu
    value: "true"
    effect: NoSchedule

# EFA support
efaEnabled: true

# Placement group for high-performance networking
placementGroup:
  groupName: p4de-placement-group
  strategy: cluster
```

```
  # Capacity Block Reservation (CBR)
  # Ensure CBR ID matches the subnet AZ with the Nodegroup subnet
  spot: false
  capacityReservation:
    capacityReservationTarget:
      capacityReservationId: "cr-abcdefghij"  # Replace with your capacity
 reservation ID
```

NVIDIA GPU Operator 使用新增至節點的標籤，nvidia.com/mig.config: "p4de-half-balanced"並以指定的設定檔分割 GPU。

6. 登入p4de執行個體。
7. 執行以下命令：

```
nvidia-smi -L
```

您應該會看到下列範例輸出：

```
[root@ip-100-64-173-145 bin]# nvidia-smi -L
GPU 0: NVIDIA A100-SXM4-80GB (UUID: GPU-ab52e33c-be48-38f2-119e-b62b9935925a)
  MIG 3g.40gb     Device  0: (UUID: MIG-da972af8-a20a-5f51-849f-bc0439f7970e)
  MIG 2g.20gb     Device  1: (UUID: MIG-7f9768b7-11a6-5de9-a8aa-e9c424400da4)
  MIG 1g.10gb     Device  2: (UUID: MIG-498adad6-6cf7-53af-9d1a-10cfd1fa53b2)
  MIG 1g.10gb     Device  3: (UUID: MIG-3f55ef65-1991-571a-ac50-0dbf50d80c5a)
GPU 1: NVIDIA A100-SXM4-80GB (UUID: GPU-0eabeccc-7498-c282-0ac7-d3c09f6af0c8)
  MIG 3g.40gb     Device  0: (UUID: MIG-80543849-ea3b-595b-b162-847568fe6e0e)
  MIG 2g.20gb     Device  1: (UUID: MIG-3af1958f-fac4-59f1-8477-9f8d08c55029)
  MIG 1g.10gb     Device  2: (UUID: MIG-401088d2-716f-527b-a970-b1fc7a4ac6b2)
  MIG 1g.10gb     Device  3: (UUID: MIG-8c56c75e-5141-501c-8f43-8cf22f422569)
GPU 2: NVIDIA A100-SXM4-80GB (UUID: GPU-1c7a1289-243f-7872-a35c-1d2d8af22dd0)
  MIG 3g.40gb     Device  0: (UUID: MIG-e9b44486-09fc-591a-b904-0d378caf2276)
  MIG 2g.20gb     Device  1: (UUID: MIG-ded93941-9f64-56a3-a9b1-a129c6edf6e4)
  MIG 1g.10gb     Device  2: (UUID: MIG-6c317d83-a078-5c25-9fa3-c8308b379aa1)
  MIG 1g.10gb     Device  3: (UUID: MIG-2b070d39-d4e9-5b11-bda6-e903372e3d08)
GPU 3: NVIDIA A100-SXM4-80GB (UUID: GPU-9a6250e2-5c59-10b7-2da8-b61d8a937233)
  MIG 3g.40gb     Device  0: (UUID: MIG-20e3cd87-7a57-5f1b-82e7-97b14ab1a5aa)
  MIG 2g.20gb     Device  1: (UUID: MIG-04430354-1575-5b42-95f4-bda6901f1ace)
  MIG 1g.10gb     Device  2: (UUID: MIG-d62ec8b6-e097-5e99-a60c-abf8eb906f91)
  MIG 1g.10gb     Device  3: (UUID: MIG-fce20069-2baa-5dd4-988a-cead08348ada)
GPU 4: NVIDIA A100-SXM4-80GB (UUID: GPU-5d09daf0-c2eb-75fd-3919-7ad8fafa5f86)
GPU 5: NVIDIA A100-SXM4-80GB (UUID: GPU-99194e04-ab2a-b519-4793-81cb2e8e9179)
GPU 6: NVIDIA A100-SXM4-80GB (UUID: GPU-c1a1910f-465a-e16f-5af1-c6aafe499cd6)
```

```
GPU 7: NVIDIA A100-SXM4-80GB (UUID: GPU-c2cfafbc-fd6e-2679-e955-2a9e09377f78)
```

NVIDIA GPU Operator 已成功將 p4de-half-balanced MIG 設定檔套用至 P4DE 執行個體，並依設定建立硬體層級 GPU 分割區。以下是分割的運作方式：

GPU Operator 從您的內嵌 MIG 設定檔套用此組態：

```
p4de-half-balanced:
  - devices: [0, 1, 2, 3]        # First 4 GPUs: MIG enabled
    mig-enabled: true
    mig-devices:
      "1g.10gb": 2               # 2x small instances (10GB each)
      "2g.20gb": 1               # 1x medium instance (20GB)
      "3g.40gb": 1               # 1x large instance (40GB)
  - devices: [4, 5, 6, 7]        # Last 4 GPUs: Full GPUs
    mig-enabled: false
```

從您的nvidia-smi -L輸出中，GPU Operator 建立的內容如下：

- 啟用 MIG 的 GPUs (0-3)：硬體分割
  - GPU 0：NVIDIA A100-SXM4-80GB
    - MIG 3g.40gb 裝置 0 – 大型工作負載 (40GB 記憶體、42 個 SMs)
    - MIG 2g.20gb 裝置 1 – 中型工作負載 (20GB 記憶體、28 個 SMs)
    - MIG 1g.10gb 裝置 2 – 小型工作負載 (10GB 記憶體、14 個 SMs)
    - MIG 1g.10gb 裝置 3 – 小型工作負載 (10GB 記憶體、14 個 SMs)
  - GPU 1：NVIDIA A100-SXM4-80GB
    - MIG 3g.40gb 裝置 0 – 相同的分割區配置
    - MIG 2g.20gb 裝置 1
    - MIG 1g.10gb 裝置 2
    - MIG 1g.10gb 裝置 3
  - GPU 2 和 GPU 3 – 與 GPU 0 和 GPU 1 相同的模式
- 完整 GPUs (4-7)：無 MIG 分割
  - GPU 4：NVIDIA A100-SXM4-80GB – 完整 80GB GPU
  - GPU 5：NVIDIA A100-SXM4-80GB – 完整 80GB GPU
  - GPU 6：NVIDIA A100-SXM4-80GB – 完整 80GB GPU

- GPU 7：NVIDIA A100-SXM4-80GB – 完整 80GB GPU

NVIDIA GPU Operator 建立 MIG 分割區後，NVIDIA DRA 驅動程式會自動偵測這些硬體隔離的執行個體，並將其用於 Kubernetes 中的動態資源配置。DRA 驅動程式會使用其特定設定檔 (1g.10gb、2g.20gb、3g.40gb) 探索每個 MIG 執行個體，並透過`mig.nvidia.com`裝置類別將其公開為可排程的資源。

DRA 驅動程式會持續監控 MIG 拓撲，並維護所有 GPUs 中可用執行個體的庫存。當 Pod 透過 請求特定 MIG 設定檔時`ResourceClaimTemplate`，DRA 驅動程式會智慧地從任何可用的 GPU 中選取適當的 MIG 執行個體，從而實現真正的硬體層級多租用戶。此動態配置可讓多個隔離的工作負載同時在相同的實體 GPU 上執行，同時維持嚴格的資源界限和效能保證。

步驟 2：測試 MIG 資源配置

現在，讓我們執行一些範例，示範 DRA 如何將 MIG 執行個體動態分配給不同的工作負載。部署`resourceclaimtemplates`和 測試 Pod，以查看 DRA 驅動程式如何將工作負載放在可用的 MIG 分割區，允許多個容器與硬體層級隔離共用 GPU 資源。

1. 建立`mig-claim-template.yaml`以包含 MIG`resourceclaimtemplates`：

```
apiVersion: v1
kind: Namespace
metadata:
  name: mig-gpu

---
# Template for 3g.40gb MIG instance (Large training)
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mig-large-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-large
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
```

```
              device.attributes['gpu.nvidia.com'].profile == '3g.40gb'

---
# Template for 2g.20gb MIG instance (Medium training)
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mig-medium-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-medium
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '2g.20gb'

---
# Template for 1g.10gb MIG instance (Small inference)
apiVersion: resource.k8s.io/v1beta1
kind: ResourceClaimTemplate
metadata:
  name: mig-small-template
  namespace: mig-gpu
spec:
  spec:
    devices:
      requests:
      - name: mig-small
        deviceClassName: mig.nvidia.com
        selectors:
        - cel:
            expression: |
              device.attributes['gpu.nvidia.com'].profile == '1g.10gb'
```

2. 套用三個範本：

```
kubectl apply -f mig-claim-template.yaml
```

3. 執行以下命令：

```
kubectl get resourceclaimtemplates -n mig-gpu
```

下列為範例輸出：

```
NAME                  AGE
mig-large-template    71m
mig-medium-template   71m
mig-small-template    71m
```

4. 建立 mig-pod.yaml 以排程多個任務，以利用此 resourceclaimtemplates：

```
---
# ConfigMap containing Python scripts for MIG pods
apiVersion: v1
kind: ConfigMap
metadata:
  name: mig-scripts-configmap
  namespace: mig-gpu
data:
  large-training-script.py: |
    import torch
    import torch.nn as nn
    import torch.optim as optim
    import time
    import os

    print(f"=== LARGE TRAINING POD (3g.40gb) ===")
    print(f"Process ID: {os.getpid()}")
    print(f"GPU available: {torch.cuda.is_available()}")
    print(f"GPU count: {torch.cuda.device_count()}")

    if torch.cuda.is_available():
        device = torch.cuda.current_device()
        print(f"Using GPU: {torch.cuda.get_device_name(device)}")
        print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
 1e9:.1f} GB")

        # Large model for 3g.40gb instance
        model = nn.Sequential(
            nn.Linear(2048, 1024),
            nn.ReLU(),
            nn.Linear(1024, 512),
```

```
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        ).cuda()

        optimizer = optim.Adam(model.parameters())
        criterion = nn.CrossEntropyLoss()

        print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

        # Training loop
        for epoch in range(100):
            # Large batch for 3g.40gb
            x = torch.randn(256, 2048).cuda()
            y = torch.randint(0, 10, (256,)).cuda()

            optimizer.zero_grad()
            output = model(x)
            loss = criterion(output, y)
            loss.backward()
            optimizer.step()

            if epoch % 10 == 0:
                print(f"Large Training - Epoch {epoch}, Loss: {loss.item():.4f}, GPU
Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(3)

        print("Large training completed on 3g.40gb MIG instance")

 medium-training-script.py: |
   import torch
   import torch.nn as nn
   import torch.optim as optim
   import time
   import os

   print(f"=== MEDIUM TRAINING POD (2g.20gb) ===")
   print(f"Process ID: {os.getpid()}")
   print(f"GPU available: {torch.cuda.is_available()}")
   print(f"GPU count: {torch.cuda.device_count()}")

   if torch.cuda.is_available():
       device = torch.cuda.current_device()
```

```
        print(f"Using GPU: {torch.cuda.get_device_name(device)}")
        print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
1e9:.1f} GB")

        # Medium model for 2g.20gb instance
        model = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        ).cuda()

        optimizer = optim.Adam(model.parameters())
        criterion = nn.CrossEntropyLoss()

        print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

        # Training loop
        for epoch in range(100):
            # Medium batch for 2g.20gb
            x = torch.randn(128, 1024).cuda()
            y = torch.randint(0, 10, (128,)).cuda()

            optimizer.zero_grad()
            output = model(x)
            loss = criterion(output, y)
            loss.backward()
            optimizer.step()

            if epoch % 10 == 0:
                print(f"Medium Training - Epoch {epoch}, Loss: {loss.item():.4f}, GPU
Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(4)

        print("Medium training completed on 2g.20gb MIG instance")

 small-inference-script.py: |
   import torch
   import torch.nn as nn
   import time
   import os

   print(f"=== SMALL INFERENCE POD (1g.10gb) ===")
```

```
    print(f"Process ID: {os.getpid()}")
    print(f"GPU available: {torch.cuda.is_available()}")
    print(f"GPU count: {torch.cuda.device_count()}")

    if torch.cuda.is_available():
        device = torch.cuda.current_device()
        print(f"Using GPU: {torch.cuda.get_device_name(device)}")
        print(f"GPU Memory: {torch.cuda.get_device_properties(device).total_memory /
 1e9:.1f} GB")

        # Small model for 1g.10gb instance
        model = nn.Sequential(
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        ).cuda()

        print(f"Model parameters: {sum(p.numel() for p in model.parameters())}")

        # Inference loop
        for i in range(200):
            with torch.no_grad():
                # Small batch for 1g.10gb
                x = torch.randn(32, 512).cuda()
                output = model(x)
                prediction = torch.argmax(output, dim=1)

                if i % 20 == 0:
                    print(f"Small Inference - Batch {i}, Predictions:
 {prediction[:5].tolist()}, GPU Memory: {torch.cuda.memory_allocated()/1e9:.2f}GB")
            time.sleep(2)

        print("Small inference completed on 1g.10gb MIG instance")

---
# Pod 1: Large training workload (3g.40gb)
apiVersion: v1
kind: Pod
metadata:
  name: mig-large-training-pod
  namespace: mig-gpu
  labels:
    app: mig-large-training
    workload-type: training
```

```
spec:
  restartPolicy: Never
  containers:
  - name: large-training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/large-training-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
      readOnly: true
    resources:
      claims:
      - name: mig-large-claim
  resourceClaims:
  - name: mig-large-claim
    resourceClaimTemplateName: mig-large-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755

---
# Pod 2: Medium training workload (2g.20gb) - can run on SAME GPU as Pod 1
apiVersion: v1
kind: Pod
metadata:
  name: mig-medium-training-pod
  namespace: mig-gpu
  labels:
    app: mig-medium-training
    workload-type: training
spec:
  restartPolicy: Never
  containers:
  - name: medium-training-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
```

```
    command: ["python", "/scripts/medium-training-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
      readOnly: true
    resources:
      claims:
      - name: mig-medium-claim
  resourceClaims:
  - name: mig-medium-claim
    resourceClaimTemplateName: mig-medium-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755

---
# Pod 3: Small inference workload (1g.10gb) - can run on SAME GPU as Pod 1 & 2
apiVersion: v1
kind: Pod
metadata:
  name: mig-small-inference-pod
  namespace: mig-gpu
  labels:
    app: mig-small-inference
    workload-type: inference
spec:
  restartPolicy: Never
  containers:
  - name: small-inference-container
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["python", "/scripts/small-inference-script.py"]
    volumeMounts:
    - name: script-volume
      mountPath: /scripts
      readOnly: true
```

```
    resources:
      claims:
      - name: mig-small-claim
  resourceClaims:
  - name: mig-small-claim
    resourceClaimTemplateName: mig-small-template
  nodeSelector:
    node.kubernetes.io/instance-type: p4de.24xlarge
    nvidia.com/gpu.present: "true"
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
  volumes:
  - name: script-volume
    configMap:
      name: mig-scripts-configmap
      defaultMode: 0755
```

5. 套用此規格，應部署三個 Pod：

```
kubctl apply -f mig-pod.yaml
```

這些 Pod 應由 DRA 驅動程式排程。

6. 檢查 DRA 驅動程式 Pod 日誌，您會看到類似以下的輸出：

```
I0717 21:50:22.925811 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:22.932499 1 driver.go:129] Returning newly prepared devices
 for claim '933e9c72-6fd6-49c5-933c-a896407dc6d1': [&Device{RequestNames:
[mig-large],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-0-
mig-9-4-4,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-0-mig-9-4-4**],}]
I0717 21:50:23.186472 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:23.191226 1 driver.go:129] Returning newly prepared devices
 for claim '61e5ddd2-8c2e-4c19-93ae-d317fecb44a4': [&Device{RequestNames:
[mig-medium],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-2-
mig-14-0-2,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-2-mig-14-0-2**],}]
I0717 21:50:23.450024 1 driver.go:87] NodePrepareResource is called: number of
 claims: 1
I0717 21:50:23.455991 1 driver.go:129] Returning newly prepared devices
 for claim '1eda9b2c-2ea6-401e-96d0-90e9b3c111b5': [&Device{RequestNames:
```

```
[mig-small],PoolName:ip-100-64-173-145.ec2.internal,DeviceName:gpu-1-
mig-19-2-1,CDIDeviceIDs:[k8s.gpu.nvidia.com/device=**gpu-1-mig-19-2-1**],}]
```

7. 驗證 resourceclaims 以查看 Pod 狀態：

```
kubectl get resourceclaims -n mig-gpu -w
```

下列為範例輸出：

```
NAME                                                STATE                AGE
mig-large-training-pod-mig-large-claim-6dpn8        pending              0s
mig-large-training-pod-mig-large-claim-6dpn8        pending              0s
mig-large-training-pod-mig-large-claim-6dpn8        allocated,reserved   0s
mig-medium-training-pod-mig-medium-claim-bk596      pending              0s
mig-medium-training-pod-mig-medium-claim-bk596      pending              0s
mig-medium-training-pod-mig-medium-claim-bk596      allocated,reserved   0s
mig-small-inference-pod-mig-small-claim-d2t58       pending              0s
mig-small-inference-pod-mig-small-claim-d2t58       pending              0s
mig-small-inference-pod-mig-small-claim-d2t58       allocated,reserved   0s
```

如您所見，所有 Pod allocated,reserved都會由 DRA 驅動程式從待定移至 。

8. nvidia-smi 從節點執行 。您會注意到三個 Python 處理器正在執行：

```
root@ip-100-64-173-145 bin]# nvidia-smi
+-----------------------------------------------------------------------------------------
+
| NVIDIA-SMI 570.158.01 Driver Version: 570.158.01 CUDA Version: 12.8 |
|-----------------------------------------+------------------------
+----------------------+
| GPU Name Persistence-M | Bus-Id Disp.A | Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap | Memory-Usage | GPU-Util Compute M. |
| | | MIG M. |
|=========================================+========================
+======================|
| 0 NVIDIA A100-SXM4-80GB On | 00000000:10:1C.0 Off | On |
| N/A 63C P0 127W / 400W | 569MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+------------------------
+----------------------+
| 1 NVIDIA A100-SXM4-80GB On | 00000000:10:1D.0 Off | On |
| N/A 56C P0 121W / 400W | 374MiB / 81920MiB | N/A Default |
| | | Enabled |
```

```
+-----------------------------------------+-----------------------
+----------------------+
| 2 NVIDIA A100-SXM4-80GB On | 00000000:20:1C.0 Off | On |
| N/A 63C P0 128W / 400W | 467MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+-----------------------
+----------------------+
| 3 NVIDIA A100-SXM4-80GB On | 00000000:20:1D.0 Off | On |
| N/A 57C P0 118W / 400W | 249MiB / 81920MiB | N/A Default |
| | | Enabled |
+-----------------------------------------+-----------------------
+----------------------+
| 4 NVIDIA A100-SXM4-80GB On | 00000000:90:1C.0 Off | 0 |
| N/A 51C P0 77W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 5 NVIDIA A100-SXM4-80GB On | 00000000:90:1D.0 Off | 0 |
| N/A 46C P0 69W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 6 NVIDIA A100-SXM4-80GB On | 00000000:A0:1C.0 Off | 0 |
| N/A 52C P0 74W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+
| 7 NVIDIA A100-SXM4-80GB On | 00000000:A0:1D.0 Off | 0 |
| N/A 47C P0 72W / 400W | 0MiB / 81920MiB | 0% Default |
| | | Disabled |
+-----------------------------------------+-----------------------
+----------------------+



+-----------------------------------------------------------------------------------------------
+
| MIG devices: |
+------------------+--------------------------------+-----------
+----------------------+
| GPU GI CI MIG | Memory-Usage | Vol| Shared |
| ID ID Dev | BAR1-Usage | SM Unc| CE ENC DEC OFA JPG |
| | | ECC| |
|==================+================================+===========
+======================|
```

```
| 0 2 0 0 | 428MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 2MiB / 32767MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 0 3 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 0 9 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 0 10 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 1 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 1 5 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 1 13 0 2 | 161MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 2MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 1 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 2 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 2 5 0 1 | 289MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 2MiB / 16383MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
| 2 13 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+-----------------------+
```

```
| 2 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+----------------------+
| 3 1 0 0 | 107MiB / 40192MiB | 42 0 | 3 0 2 0 0 |
| | 0MiB / 32767MiB | | |
+------------------+----------------------------------+-----------
+----------------------+
| 3 5 0 1 | 71MiB / 19968MiB | 28 0 | 2 0 1 0 0 |
| | 0MiB / 16383MiB | | |
+------------------+----------------------------------+-----------
+----------------------+
| 3 13 0 2 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+----------------------+
| 3 14 0 3 | 36MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| | 0MiB / 8191MiB | | |
+------------------+----------------------------------+-----------
+----------------------+



+-----------------------------------------------------------------------------------------
+
| Processes: |
| GPU GI CI PID Type Process name GPU Memory |
| ID ID Usage |
|
=========================================================================================|
**| 0 2 0 64080 C python 312MiB |
| 1 13 0 64085 C python 118MiB |
| 2 5 0 64073 C python 210MiB |**
+-----------------------------------------------------------------------------------------
+
```

使用 GB200 P6e 執行個體透過 IMEX 最佳化 GPU 工作負載

IMEX (Internode Memory Exchange) 可讓跨節點的記憶體一致性通訊，以便在 NVIDIA GB200 UltraServers 上進行分散式訓練。

執行下列步驟。

1. 使用名為 的檔案為ComputeDomain多節點訓練定義 imex-compute-domain.yaml：

```
apiVersion: resource.nvidia.com/v1beta1
kind: ComputeDomain
metadata:
  name: distributed-training-domain
  namespace: default
spec:
  numNodes: 2
  channel:
    resourceClaimTemplate:
      name: imex-channel-template
```

2. 使用 IMEX 頻道搭配名為 的檔案來定義 Podimex-pod.yaml：

```
apiVersion: v1
kind: Pod
metadata:
  name: imex-distributed-training
  namespace: default
  labels:
    app: imex-training
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: nvidia.com/gpu.clique
            operator: Exists
  containers:
  - name: distributed-training
    image: nvcr.io/nvidia/pytorch:25.04-py3
    command: ["bash", "-c"]
    args:
    - |
      echo "=== IMEX Channel Verification ==="
      ls -la /dev/nvidia-caps-imex-channels/
      echo ""

      echo "=== GPU Information ==="
      nvidia-smi
      echo ""
```

```
      echo "=== NCCL Test (if available) ==="
      python -c "
      import torch
      import torch.distributed as dist
      import os

      print(f'CUDA available: {torch.cuda.is_available()}')
      print(f'CUDA device count: {torch.cuda.device_count()}')

      if torch.cuda.is_available():
          for i in range(torch.cuda.device_count()):
              print(f'GPU {i}: {torch.cuda.get_device_name(i)}')

      # Check for IMEX environment variables
      imex_vars = [k for k in os.environ.keys() if 'IMEX' in k or 'NVLINK' in k]
      if imex_vars:
          print('IMEX Environment Variables:')
          for var in imex_vars:
              print(f'  {var}={os.environ[var]}')

      print('IMEX channel verification completed')
      "

      # Keep container running for inspection
      sleep 3600
    resources:
      claims:
      - name: imex-channel-0
      - name: imex-channel-1
  resourceClaims:
  - name: imex-channel-0
    resourceClaimTemplateName: imex-channel-template
  - name: imex-channel-1
    resourceClaimTemplateName: imex-channel-template
  tolerations:
  - key: nvidia.com/gpu
    operator: Exists
    effect: NoSchedule
```

Note

這需要 P6e GB200 執行個體。

3. 透過套用 ComputeDomain和 範本來部署 IMEX：

```
kubectl apply -f imex-claim-template.yaml
kubectl apply -f imex-compute-domain.yaml
kubectl apply -f imex-pod.yaml
```

4. 檢查ComputeDomain狀態。

```
kubectl get computedomain distributed-training-domain
```

5. 監控 IMEX 協助程式部署。

```
kubectl get pods -n nvidia-dra-driver -l resource.nvidia.com/computeDomain
```

6. 檢查 Pod 中的 IMEX 頻道：

```
kubectl exec imex-distributed-training -- ls -la /dev/nvidia-caps-imex-channels/
```

7. 檢視 Pod 日誌：

```
kubectl logs imex-distributed-training
```

   以下是預期輸出的範例：

```
=== IMEX Channel Verification ===
total 0
drwxr-xr-x. 2 root root 80 Jul 8 10:45 .
drwxr-xr-x. 6 root root 380 Jul 8 10:45 ..
crw-rw-rw-. 1 root root 241, 0 Jul 8 10:45 channel0
crw-rw-rw-. 1 root root 241, 1 Jul 8 10:45 channel1
```

如需詳細資訊，請參閱 GitHub 上的 NVIDIA 範例。

CPU 推論和協同運作

> **Tip**
>
> 註冊即將舉行的 Amazon EKS AI/ML 研討會。

概觀

CPU 執行個體是 Amazon EKS 上各種 AI 工作負載的一級運算選項。從小型語言模型 (SLMs) 和傳統 ML 推論到資料管道和代理程式協同運作，CPUs 提供強大的價格效能、廣泛的容量可用性和熟悉的 Kubernetes 排程語意。

CPU 和 GPU 是互補的，而不是競爭性。隨著代理式 AI 管道的複雜性增加，CPU 工作負載表面也會隨之增加：每個推論呼叫都被工具執行、內容組合、向量搜尋、護欄和協同運作邏輯包圍，這些邏輯全都在 CPU 上執行。我們建議刻意設計使用這兩種運算類型的架構，將每個工作負載放在提供最佳成本效能的 層。

並非每個工作負載都需要 GPU。路由、分類、擷取、內嵌、協同運作，以及不斷增加的語言模型推論份額，全都在 CPU 上有效執行。跨 arm64 和 x86 的新一代 CPU 執行個體為 ML 推論提供強大的價格效能。結合 Karpenter 的節點整合、KEDA 的事件驅動擴展和量化模型服務，這提供了一個生產就緒的堆疊，平台團隊無需深厚的 GPU 專業知識即可操作。

本指南適用於：

- 為 AI 工作負載設計多租用戶 EKS 叢集的平台工程師。
- ML 從業人員評估 30B 參數下模型的推論後端。
- FinOps 團隊在不犧牲 SLOs 的情況下尋找具體的成本控制桿。

您將學到什麼：

- 哪些 AI 工作負載屬於 CPUs以及需要 GPUs 或 Trainium 的位置。
- 如何將四維決策架構套用至任何新的工作負載。
- 兩種生產模式：代理式 SLM 預先篩選和高密度模型陣列。
- 最佳化最佳實務：量化、bin-packing、Spot 排程和自動擴展。

> **Warning**
>
> 本指南中的每個建議都應以經驗驗證。正確的執行個體系列 (arm64、x86、GPU 或 Trainium) 取決於您的模型、資料和延遲預算。使用本指南作為明智的起點，然後在遞交之前進行基準測試。

為什麼選擇適用於 AI 工作負載的 CPUs？

生產 AI 管道會將工作分散到許多運算層。CPUs處理路由、分類、擷取、協同運作，以及越來越多的推論。新一代 CPU 執行個體提供強大的價格效能和熟悉的 Kubernetes 排程，使它們成為許多 AI 工作負載的實用選項。

讓 CPU 吸引這些工作負載的三個因素：

容量可用性

GPU 執行個體經常需要提前幾週保留容量。CPU 執行個體可在所有 AWS 區域中廣泛使用，無需特殊化裝置外掛程式、DRA 組態，也無需 MIG 分割。當您需要快速擴展時，CPU 容量是最容易使用的選項。

經濟效益

新一代 CPU 執行個體為 ML 推論提供強大的價格效能。對於執行 FinOps 檢閱或管理多租用戶叢集的團隊而言，CPU 和 GPU 之間的成本差異很重要，尤其是 GPU 加速提供降低回報SLMs。我們建議跨可用執行個體系列 (Graviton、AMD、Intel) 進行基準測試，以尋找工作負載的最佳cost-per-token。

操作簡單性

CPU Pod 使用標準 Kubernetes 排程 (`requests`、`limits`、節點親和性、拓撲分散）。沒有裝置外掛程式、沒有自訂排程器、沒有`nvidia.com/gpu`資源類型。想要在沒有深度 GPU 專業知識的情況下執行 AI 工作負載的團隊可以更快地在 CPU 上實現生產。

在代理管道中增長 CPU 表面

在代理式 AI 管道中，每個 GPU 推論呼叫都被 CPU 工作包圍：工具執行、內容組件、向量搜尋、內嵌查詢、護欄、回應驗證、記憶體管理和協同運作邏輯。隨著客服人員的成長更複雜（工具越多、鏈結越長、多步驟推理），這些 CPU 工作負載會以超線性方式成長。例如 MCP（模型內容通訊協定）的通訊協定會進一步擴增：每個 MCP 工具呼叫都會觸發完全在 CPU 上執行的資料擷取、轉換和格式化。

CPU 與 GPU/ Trainium：何時選擇每個

Factor	選擇 CPU	選擇 GPU/Trainium
模型大小	SLMs1-8B（量化）；內嵌；分類器	8B+ 用於低延遲線上推論；一般為 70B+

Factor	選擇 CPU	選擇 GPU/Trainium
延遲 SLO	可接受 p95 100-500ms	需要 p95 < 50 毫秒
並行	每個端點 < 100 要求/秒	> 100 請求/秒持續
工作負載類型	協調、擷取、ETL、批次評分	線上推論、微調、訓練
Capacity	立即可用性，無保留	通常需要預留容量
成本敏感度	CPU 為符合資格的工作負載提供最佳的 $/Token	GPU 在高使用率下攤銷
團隊專業知識	標準 Kubernetes 操作	需要 GPU 操作知識
資料主權	VPC 中的 SLM 推論；完整稽核追蹤；資料永遠不會離開您的帳戶	如果自我管理則相同；不適用於外部 APIs

> **Tip**
>
> 這些閾值是起點。建議您在候選執行個體系列 (arm64 和 x86) 上執行目標推論引擎，並搭配實際模型和流量模式，再遞交至運算層。

工作負載決策架構

選擇適合 AI 工作負載的運算，有四個維度：

1. 模型大小和精確度：量化是否將品質保持在可接受的範圍內？
2. 延遲和輸送量 SLOs：您的 p50/p95 目標和尖峰請求率是多少？
3. 工作負載類型：線上推論、批次評分、擷取或協同運作？
4. 成本和容量限制：FinOps 預算、區域可用性、保留策略？

使用下表做為決策矩陣。

工作負載	CPU	GPU / Trainium	備註
SLMs(1-8B 參數，量化）	預設選擇。100-500ms 延遲、中等 QPS 時的強大價格優惠。跨執行個體系列的基準。	當 p95 <50ms 或並行 >100 要求/秒時。	建議 Q4_K_M 或 Q8_0 量化
中型模型 (8-30B 參數）	批次、非同步、離線評分。測試 Q4 量化。	線上推論、長內容、緊密延遲。	跨執行個體系列的基準 Q4
大型 LLMs (70B+ 參數）	非Non-real-time、大量量化	生產線上推論的預設值	即使 70B 也可以在 CPU 上執行；預期高延遲
傳統 ML/嵌入/CV	高密度服務；跨節點的 bin-pack。	大規模複視或多模態。	TorchServe，CPU 上的 Triton 處理數千個模型。
資料管道/ETL/合成資料	CPU 上的 Ray 和 Spark 用於資料準備和功能工程。	N/A	CPUs錨定整個資料準備階段
代理程式協同運作/RAG 擷取	網路繫結服務：API 閘道、代理層、擷取器、區塊。	N/A	高頻寬 CPU 執行個體的優點。
微調/訓練	資料準備和管道協調。	模型訓練和分割。	混合：CPU 準備、GPU 訓練、CPU 推論。
合規敏感推論 (FSI、醫療保健、政府）	CPU 上 VPC SLMs。資料會保留帳戶內的完整稽核線索。	在 GPU 上自我管理時相同。	CPU 在受管制環境中獲得 sub-8B模型的成本。

Warning

雖然在技術上可以在具有大量量化 (Q4 或更低） 的 CPU 上執行 70B+ 模型，但這僅適用於 non-real-time、離線或批次工作負載。預期的字符產生率為低個位數 (1-5 個字符/秒）、即使在

第 4 Q4也超過 40GB 的記憶體需求，以及長輸出的每個回應的延遲，以分鐘為單位。對於任何互動式或延遲敏感的使用案例，70B+ 模型屬於 GPU 或 Trainium。

快速基準工作流程

在遞交執行個體系列之前，我們建議在單一可比較指標上，執行比較候選 CPU 系列 (arm64 和 x86) 與 GPU 的結構化基準測試：在目標 p95 延遲時每 1，000 個查詢成本。使用相同的模型組態（相同量化、內容大小、執行緒計數）、負載測試每個系列部署一個節點，並進行比較。如果 CPU 執行個體符合您的 p95 SLO，它可能會降低成本。如果略微遺漏，請先嘗試該系列中最新一代的 ，再移至 GPU。如果並行目標的延遲仍然太高，則表示將工作負載移至 GPU 的訊號。

生產模式

模式 1：代理式 AI — 使用 LLM 升級的 CPU 上的 SLM 預先篩選

大多數客服人員工作流程會重複執行相同的縮小模式：分類請求、挑選工具、擷取結構化資料、驗證回應。這些任務不需要 70B 參數模型。

對 SLMs(arXiv：2506.02153) 的研究顯示，在 10B 參數以下的模型專門用於網域時，可以在受限子任務上比對或超過大型 LLMs，同時以顯著較低的成本和延遲有效率地在 CPU 上執行。針對特定網域微調模型時，其佔用空間越小，它比叫用一般用途 LLM 更準確、更便宜。

在此模式中，CPU 上的 SLM 會end-to-end請求。路由層（也在 CPU 上執行）只會將真正複雜的案例升級為 GPU 託管的 LLM。

在 CPU 上執行的元件：

- API 閘道/代理層 — 處理身分驗證、路由、速率限制
- 客服人員協調人員 — 管理工具呼叫和狀態
- SLM 推論服務 — 在 CPU 上使用 llama.cpp、Ollama 或 vLLM 等推論引擎來量化 1-8B 模型
- 向量擷取 — CPU 節點上的 OpenSearch

GPU/Trainium 上的元件：

- 適用於複雜合成、長內容推理的大型 LLM

此模式運作的原因：在許多代理工作流程中，60-80% 的請求可由 SLM 分類或擷取。對於您避免的每個 LLM 呼叫，您也可以避免組裝大型內容視窗、對長回應執行護欄和管理複雜狀態的周邊 CPU 工作。CPU 層會獨立於 GPU 層擴展。

典型代理管道中的 CPU 工作負載類別包括：工具執行 (MCP 伺服器呼叫、API 呼叫、資料庫查詢）、內容組件、向量搜尋和內嵌查詢、協調和規劃邏輯、護欄和安全篩選、回應驗證和格式化、代理記憶體和狀態管理，以及記錄/可觀測性。

此模式也適合微調生命週期：收集 CPU 節點上的網域資料、微調 GPU，然後將量化模型部署回 CPU 以進行推論，成本遠低於 LLM 端點。LoRA Land (arXiv：2405.00732) 的研究顯示，經過微調的 7B 模型在大多數測試的網域特定任務上都優於 GPT-4，使得「微調小模型並在 CPU 上執行」路徑在許多生產使用案例中可行。

模式 2：高密度 CPU 模型陣列

生產 ML 管道會定期部署數百或數千個較小的模型：內嵌、推薦者、分類器、以 BERT 為基礎的評分者和電腦視覺模型。個別輕量型，這些模型會在每個模型指派自己的 GPU 資源時變得昂貴。

我們建議使用高密度 CPU 服務（在 CPU 上使用 TorchServe 或 Triton 來封裝每個節點的多個模型），並對觀察到的負載進行 Karpenter 管理節點生命週期和 KEDA 擴展。

此模式自然延伸至 RAG 架構：從 OpenSearch 內嵌產生、文件區塊化和擷取都會以經濟實惠的方式在 CPU 節點上執行，僅在最終產生步驟將結果饋送至 GPU 託管的 LLM。CPU 陣列會處理磁碟區；GPU 會處理複雜性。

對於受監管的產業（金融服務、醫療保健、政府），此模式特別吸引人：數百種在 CPU 上於 VPC 中執行的特殊模型，其中包含從不離開帳戶的完整稽核線索和資料。自我管理推論的合規要求自然符合 sub-8B以下模型 CPU 的成本優勢。

最佳化最佳實務

量化

在 CPU 上以完整的 BF16 執行 7B 模型並不切實際；在 Q4 量化時執行模型是可行且符合成本效益的。了解為什麼量化有助於 CPU，是做出良好基礎設施決策的關鍵。

為什麼量化對 CPU 推論很重要。CPU 推論是記憶體頻寬繫結，而不是運算繫結。在解碼階段（一次產生一個權杖）期間，系統會針對產生的每個權杖，從 RAM 讀取模型的整個權重。CPU 會花費大部分時間等待資料從記憶體抵達，而不是進行數學運算。BF16 的 7B 模型大約是 14GB；在 Q4_K_M，它縮減到大約 4GB。由於瓶頸正在將位元組從 RAM 移動到 CPU 核心，因此 3.5 倍較小的模型讀取速

度快 3.5 倍，幾乎直接轉換為 3.5 倍的字符產生速度。這就是為什麼量化是 CPU 推論最有影響力的單一最佳化，以及為什麼具有更多記憶體通道的較新 CPU 世代即使在相同的時鐘速度下也能產生更快的推論。

我們建議您使用架構最佳化後端建置推論引擎 (ARM NEON/SVE2 for arm64、AVX-512/AMX for x86)、設定執行緒計數等於 vCPU 計數，以及選取 Q4_K_M 或 Q8_0 量化格式。

量化	品質影響	輸送量與 BF16	使用案例
Q4_K_M	低 (1-3% 的多工差異，取決於模型）	~4-5 倍速度	SLMs生產預設值
Q8_0	可忽略	~2 倍快	對品質敏感的任務
Q5_K_M	非常低	~3.5 倍	品質和速度的平衡
BF16	無	1x （基準）	避免在 CPU 上使用 7B+ 模型

對於 sub-2B模型，CPU 在價格效能與 GPU 方面獲勝。這些模型非常小，足以讓 GPU 加速提供最低效益，同時每小時成本也明顯較高。如果您的工作負載可以使用 sub-2B 模型，則建議使用 CPU。

架構特定的最佳化：在 arm64 上，目前世代的 Graviton 執行個體支援 SVE2。為您的目標建置具有適當`-march`旗標的推論引擎。在 x86 上，AMD EPYC 執行個體支援 AVX-512，而 Intel Xeon 執行個體新增 AMX 進行矩陣加速。由於推論是記憶體頻寬繫結，因此具有更多 DDR5 記憶體通道的較新 CPU 世代即使在相同的時鐘速度下也能產生更快的推論。選擇執行個體類型時，請優先考慮核心計數的記憶體頻寬。

內容視窗大小：對於分類和路由工作負載，輸入通常低於 200 個字符，而輸出為 2-3 個字符。設定小型內容視窗 （例如 512 權杖） 而非預設的 2048 可減少 KV 快取記憶體用量，並改善每個請求的延遲。只有在輸入真正長時，才增加內容視窗。

Flash 注意：如果您的推論引擎支援，請啟用 Flash 注意。Flash Attention 可避免完全注意力矩陣具體化，以減少注意力運算的記憶體使用量。在 CPU 上，優點小於 GPU，但仍有助於更長的輸入。

> **Tip**
>
> Q4_K_M 品質降級因模型和任務而異。部署到生產環境之前，請務必先評估自己的資料集。

密集服務的 Bin-packing

對於傳統 ML 和內嵌模型 （通常每個模型 <500MB)，目標是在穩定的尾部延遲下每個節點的最大 Pod 密度。兩件事會決定您是否實現這一點：準確的資源請求和受控執行緒。

`requests` 以實際負載下觀察到的 p50-p90 用量為基礎。使用負載測試中的金絲雀、VPA 建議或 Prometheus 長條圖。兩個方向的預設值幾乎一律錯誤。

ML 程式庫 (PyTorch、ONNX 執行期、MLK、OpenBLAS) 會在節點上看到 vCPUs，而不是配置給 Pod 的 CPUs，產生任意數量的執行緒。在具有 20 個 Pod 的密集節點上，每個 Pod 都會嘗試產生 32 個執行緒。節點在內容切換和 p99 延遲峰值時悄悄衝動。明確修正此問題：

```
env:
  - name: OMP_NUM_THREADS
    value: "2"          # match your cpu request (2000m = 2 threads)
  - name: MKL_NUM_THREADS
    value: "2"
  - name: OPENBLAS_NUM_THREADS
    value: "2"
  - name: INTRA_OP_NUM_THREADS    # PyTorch / ONNX Runtime
    value: "2"
  - name: NUM_INTER_THREADS
    value: "1"          # keep inter-op parallelism minimal
```

將每個值設定為等於或低於您的 CPU 請求。對於具有 4 個以上核心的 Pod，基準測試從 2-4 個執行緒開始。由於快取效率，許多小型模型的效能更好，執行緒更少。如果您使用 HPA 搭配許多精簡型 Pod，則每個 Pod 幾乎一律會獲勝 1-2 個執行緒。

排程和成本最佳化

兩種實務可大幅降低 CPU 推論成本：具有 Karpenter 整合的 Spot 執行個體，以及多封存容器映像。

Karpenter 的整合非常適合 CPU 推論，因為佇列或負載平衡器後方的無狀態推論 Pod 可正常容忍中斷。設定整合以對預算限制並行中斷的未充分利用節點採取行動 （例如，一次 20% 的節點），以避免在縮減規模期間容量下降。Karpenter 的`nodePool`規格可讓您在單一集區中混合 Spot 和隨需容量，並以 Spot 做為偏好選項，並以隨需做為備用。

建置多封存映像 (`arm64` 和 `amd64`) 會進一步解除鎖定。有了這兩種架構，Karpenter 可以根據即時價格和可用性，從完整的執行個體系列 (Graviton、AMD、Intel) 中進行選擇。這對於跨執行個體類型和架構多樣化降低中斷頻率的 Spot 工作負載特別有用。將 `docker buildx`或 CI 管道與多平台建置搭配使用，以產生涵蓋兩種架構的單一資訊清單。

容器啟動最佳化

當 Karpenter 佈建新節點（向上擴展、Spot 替換）時，容器執行時間需要提取推論映像，Pod 才能啟動。對於多 GB 推論映像，這可能會為 Pod 啟動增加 30-60 秒。

我們建議您使用 Bottlerocket 做為推論工作負載的節點作業系統，並在平行提取/解壓縮模式中結合 SOCI 快照器。SOCI 會以平行區塊型下載取代預設的循序層擷取，大幅縮短大型影像的影像提取時間。您不需要變更容器映像。

如需詳細的組態指引，請參閱本指南的效能一節，其中涵蓋 SOCI 組態、EBS 快照預先提取和容器執行期快取策略。

可觀測性

如果模型層沒有可觀測性，您會以盲目方式擴展。我們建議為每個推論服務公開 Prometheus 指標，並使用它們來驅動 KEDA 擴展和操作儀表板。

大多數推論伺服器 (llama.cpp、vLLM、Triton、TorchServe) 會在`/metrics`端點公開 Prometheus 相容指標。指標名稱因伺服器而異，但概念相同。

要檢測的關鍵指標：

指標類別	說明	警示閾值
請求處理/處理中	伺服器目前正在處理的請求數量。	使用 進行擴展（請參閱下面的自動擴展一節）
已排入佇列/延遲的請求	等待處理槽的請求數量。	擴展觸發。任何持續佇列表示延遲即將降低。
字符輸送量	每秒產生的字符數。	如果輸送量低於負載基準的 50%，請發出提醒
請求延遲	End-to-end延遲長條圖（提示處理 + 字符產生）。	p95 超過 SLO 的提醒
KV 快取使用率	金鑰值快取的完整程度 (0.0 到 1.0)。接近 1.0 表示伺服器將開始拒絕或排入佇列請求。	警示超過 85%

指標類別	說明	警示閾值
容器記憶體	每個 Pod 的 RSS 記憶體 (container_memory_working_set_bytes)。	警示為限制的 85%

自動擴展：根據佇列深度進行擴展，而非 CPU 使用率

CPU 使用率是飽和指標。在延遲已降低之後，它會遽增。在以使用率為基礎的自動調整規模反應時，使用者已經在等待。

佇列深度（請求延遲/等待）是主要指標。它會在延遲降低之前上升，因為當所有處理插槽忙碌時，請求會開始排入佇列。擴展佇列深度表示在現有複本仍正常回應時佈建新的複本。

KEDA 支援使用 將多個指標合併為單一擴展公式 `scalingModifiers`（需要 KEDA 2.12+)。推論工作負載的建議模式是將處理中的請求與排入佇列的請求結合，大幅加權佇列指標：

```
advanced:
  scalingModifiers:
    formula: "running + (waiting * 10)"
    target: "25"
    activationTarget: "5"
```

公式`running + (waiting * 10)`表示即使有 3 個佇列請求，也會將合併指標推送至 55，遠高於目標 25。擴展會在延遲降低之前開始。5 `activationTarget`的 可防止雜訊觸發不必要的scale-from-zero事件。

評估 CPU 優先工作負載的模型品質

在 CPU 上部署量化 SLM 是一項成本和延遲決策。只有在模型仍為您的工作負載產生正確、有用的輸出時才有意義。

較小的模型或量化會降低運算成本，但可以降低品質。影響會有所不同。在 CPU 上運作良好的工作負載（分類、擷取、路由、摘要、內嵌）通常會在 3B-7B 範圍內保持良好的品質，並具有適當的量化和提示。

要評估的內容

不同的工作負載會以不同的方式降級：

工作負載	哪些項目可能會降級	要測量的項目
意圖或票證分類	輸入不明確時發生錯誤	準確度，每個類別 F1
結構化擷取 (JSON)	缺少欄位或結構描述錯誤	完全相符，結構描述有效性
RAG 答案	幻覺或忽略內容	誠實、回答相關性
摘要	缺少事實或涵蓋範圍不佳	ROUGE-L、BERTScore、人工審核
客服人員轉接	選取錯誤的工具	工具準確度
嵌入項目	較差的擷取品質	Recall@K、NDCG

實際的評估工作流程

我們建議您在生產前建立品質檢查，類似於在選擇執行個體類型之前執行負載測試的方式。工作流程有四個階段：

1. 建置評估資料集：從實際工作負載建置小型評估資料集 (100-300 個標籤範例）。避免測量一般推理而非真實任務的一般基準，例如 MMLU。
2. 建立基準 — 根據信任的模型執行資料集（例如，您知道的大型 LLM 會產生正確的結果）。
3. 測試 CPU 模型 — 在您的量化 SLM 上執行相同的資料集並進行比較。
4. 評估 — 在測試之前定義品質閾值，例如「SLM 準確度在基準的 5 個百分點內」。正確的閾值取決於任務：人工檢閱的分類器可以容忍比系統進行自動決策更多的錯誤。

如何復原品質

如果模型效能不佳，請嘗試這些動作：

- 在提示中新增幾個拍攝範例：零成本，立即。在提示中包含 3-5 個已標記的範例，通常會縮小分類和擷取任務的差距。
- 使用更高品質的量化格式：從Q4 移至Q8 通常會以大約 2 倍的記憶體和更低的輸送量來還原大部分遺失的品質。
- 使用混合路由：讓 SLM 處理簡單的案例，並將困難的輸入傳送至較大的模型。這是架構變更，但大多數流量的 CPU 成本都很低。

- 微調網域上的模型：最昂貴的選項，但最有效。來自 LoRA Land (arXiv：2405.00732) 的研究發現，微調後的 7B 模型在大多數測試的網域特定任務上都優於 GPT-4。

參考

- Amazon EKS 文件
- AWS Graviton 開發人員指南
- Karpenter 文件
- KEDA 文件
- llama.cpp
- vLLM 文件
- 奧拉馬
- Ray 服務文件
- Amazon OpenSearch Service
- AWS Spot 最佳實務
- LoRA Land：效能優於 GPT-4 的微調開放原始碼 LLMs
- 模型內容通訊協定 (MCP) 規格
- 客服人員 AI (NVIDIA) 的小型語言模型
- SOCI 快照
- Bottlerocket 作業系統
- AI on EKS：容器啟動最佳化

聯網

> **Tip**
>
> 註冊即將舉行的 Amazon EKS AI/ML 研討會。

對於具有高節點間通訊的應用程式，請考慮較高的網路頻寬或彈性布料轉接器

對於 Amazon EKS 上具有高節點間通訊需求的分散式訓練工作負載，請考慮選取具有較高網路頻寬或 Elastic Fabric Adapter (EFA) 的執行個體。網路效能不足會造成資料傳輸的瓶頸，從而減緩分散式多 GPU 訓練等機器學習任務。請注意，推論工作負載通常不會有高節點間通訊。

範例

例如，使用 Karpenter：

```
apiVersion: v1
kind: Pod
metadata:
  name: ml-workload
spec:
  nodeSelector:
    karpenter.k8s.aws/instance-network-bandwidth: "100000"  # 100 Gbps in Mbps
    node.kubernetes.io/instance-type: p5.48xlarge  # EFA-enabled instance
  containers:
  - name: training-job
    image: `763104351884.dkr.ecr.us-west-2.amazonaws.com/pytorch-inference:2.6.0-gpu-
py312-cu124-ubuntu22.04-ec2-v1.6`
    resources:
      limits:
        vpc.amazonaws.com/efa: 1  # Requires EFA device plugin
```

確保 MPI 和 NCCL 等工具已安裝在您的容器映像中，以利用 EFA 進行訓練任務。

在大型 GPU 執行個體上規劃 IP 地址使用

根據預設，Amazon VPC CNI 外掛程式會預先配置 IP 地址，以確保可以快速排程 Pod，保持連接一個完整的備用 ENI，並填入 IPs。在大型執行個體上，這可能會導致每個節點預留數十個 IPs，即使只有幾個 Pod 執行中。

此不相符項目常見於訓練和推論工作負載，其中每個節點的 Pod 密度較低。在叢集擴展時，特別是在啟動多個 GPU 節點且每個 GPU 節點各有幾個 Pod 的自動擴展事件期間，這可能會導致子網路 IP 耗盡，即使實際 IP 使用率很低。

若要緩解這種情況，請調校 `WARM_IP_TARGET`、 `MINIMUM_IP_TARGET`和 `WARM_ENI_TARGET`變數，以符合您的實際 Pod 密度。VPC CNI 的 ENI 和 IP 目標設定的詳細資訊。

如需最佳化 IP 消耗的完整指南，請參閱最佳化 IP 地址使用率。

安全

> **Tip**
>
> 註冊即將舉行的 Amazon EKS AI/ML 研討會。

安全與合規

考慮使用具有 KMS 的 S3 進行加密合規儲存

除非您另有指定，否則所有 S3 儲存貯體預設都會使用 SSE-S3 來加密靜態物件。不過，您可以選擇設定儲存貯體，改為使用伺服器端加密搭配 AWS Key Management Service (AWS KMS) 金鑰 (SSE-KMS)。AWS KMS 中的安全控制可以協助您符合加密相關合規要求。您可以使用這些 KMS 金鑰來保護 Amazon S3 儲存貯體中的資料。當您搭配 S3 儲存貯體使用 SSE-KMS 加密時，AWS KMS 金鑰必須與儲存貯體位於相同的區域。

將一般用途儲存貯體設定為使用 S3-KMS 的 S3 儲存貯體金鑰，透過減少從 Amazon S3 到 AWS KMS 的請求流量，將 AWS KMS 請求成本降低高達 99%。目錄儲存貯體中的 `GET`和 `PUT`操作一律會啟用 S3 儲存貯體金鑰，且無法停用。

請注意，Amazon S3 Express One Zone 使用稱為 S3 目錄儲存貯體的特定儲存貯體類型。目錄儲存貯體僅適用於 S3 Express One Zone 儲存類別，並啟用高效能、低延遲的存取。若要在 S3 目錄儲存貯體上設定預設儲存貯體加密，請使用 AWS CLI，並指定 KMS 金鑰 ID 或 ARN，而非別名，如下列範例所示：

Example

```
aws s3api put-bucket-encryption --bucket my-directory-bucket --server-side-encryption-
configuration \
   '{"Rules": [{"ApplyServerSideEncryptionByDefault": {"SSEAlgorithm": "aws:kms",
 "KMSMasterKeyID": "1234abcd-12ab-34cd-56ef-1234567890ab"}}]}'
```

確保您的 EKS Pod 的 IAM 角色具有存取加密物件的 KMS 許可（例如 `kms:Decrypt`)。在預備環境中進行測試，方法是將範例模型上傳到儲存貯體、將其掛載在 Pod 中（例如，透過掛載點 S3 CSI 驅動程式），並確認 Pod 可以讀取加密的資料而不會發生錯誤。透過 AWS CloudTrail 稽核日誌，以確認是否符合加密要求。如需設定詳細資訊和金鑰管理，請參閱 KMS 文件。

儲存

> **Tip**
>
> 註冊即將舉行的 Amazon EKS AI/ML 研討會。

資料管理和儲存

使用 CSI 驅動程式將 AI 模型部署至 Pod

AI/ML 工作負載通常需要存取大型模型成品（例如，經過訓練的權重、組態），而 Pod 需要可靠、可擴展的方式來存取這些成品，而無需將其內嵌在容器映像中，這可能會增加影像大小和容器登錄提取時間。為了降低管理磁碟區掛載的操作開銷，我們建議您使用各自的 CSI 驅動程式，將 Amazon 儲存服務（例如 S3、FSx for Lustre、FSx for OpenZFS、EFS) 掛載為持久性磁碟區 (PVs)，以將 AI 模型部署至 Pod。如需實作詳細資訊，請參閱本節中的後續主題。

在 EKS 上最佳化 ML 模型快取的儲存體

利用最佳儲存解決方案對於將 Pod 和應用程式啟動延遲降至最低、減少記憶體使用量、取得所需的效能層級以加速工作負載，以及確保 ML 工作負載的可擴展性至關重要。ML 工作負載通常依賴模型檔案（權重），這些檔案可能很大，需要跨 Pod 或節點共用存取資料。選取最佳儲存解決方案取決於工作負載的特性，例如單一節點效率、多節點存取、延遲需求、成本限制，以及資料整合需求（例如使用 Amazon S3 資料儲存庫）。我們建議您使用工作負載對不同的儲存解決方案進行基準測試，以了解哪個解決方案符合您的需求，而且我們提供下列選項，協助您根據您的工作負載需求進行評估。

EKS CSI 驅動程式支援下列 AWS Storage 服務，每個服務都有自己的 CSI 驅動程式，並為 AI 和 ML 工作流程提供自己的優勢：

- Amazon S3 掛載點
- Amazon FSx for Lustre
- Amazon FSx for OpenZFS
- Amazon EFS
- Amazon EBS

AWS Storage 服務的選擇取決於您的部署架構、規模、效能需求和成本策略。儲存 CSI 驅動程式需要安裝在 EKS 叢集上，這可讓 CSI 驅動程式在 Pod 生命週期外建立和管理持久性磁碟區 (PV)。使用

CSI 驅動程式，您可以將支援的 AWS Storage 服務的 PV 定義建立為 EKS 叢集資源。然後，Pod 可以透過為 PV 建立持久性磁碟區宣告 (PVC) 來存取其資料磁碟區的這些儲存磁碟區。視 AWS 儲存服務和部署案例而定，單一 PVC （及其相關聯的 PV) 可以連接到工作負載的多個 Pod。例如，對於 ML 訓練，共用訓練資料存放在 PV 上，並由多個 Pod 存取；對於即時線上推論，LLM 模型會快取在 PV 上，並由多個 Pod 存取。AWS Storage 服務的範例 PV 和 PVC YAML 檔案提供如下，以協助您開始使用。

監控效能 磁碟效能不佳可能會延遲容器映像讀取、增加 Pod 啟動延遲，以及降低推論或訓練輸送量。使用 Amazon CloudWatch 監控 AWS 儲存服務的效能指標。當您發現效能瓶頸時，請修改儲存組態參數以最佳化效能。

案例：多個 GPU 執行個體工作負載

Amazon FSx for Lustre：如果您有多個 EC2 GPU 運算執行個體環境具有延遲敏感和高頻寬輸送量動態工作負載，例如分散式訓練和模型服務，而且您需要原生 Amazon S3 資料儲存庫整合，我們建議您使用 Amazon FSx for Lustre。FSx for Lustre 提供全受管的高效能平行檔案系統，專為高效能運算 (HPC)、Machine Learning等運算密集型工作負載而設計。

您可以安裝 FSx for Lustre CSI 驅動程式，將 FSx 檔案系統掛載為持久性磁碟區 (PV)，然後將 FSx for Lustre 檔案系統部署為獨立高效能快取，或部署為 S3-linked檔案系統，以做為 S3 資料的高效能快取，為 GPU 運算執行個體的資料存取提供快速 I/O 和高輸送量。FSx for Lustre 可以使用 Scratch-SSD 或 Persistent-SSD 儲存選項進行部署：

- 暫存 SSD 儲存：建議用於暫時性或短期 （小時） 的工作負載，並佈建每個 TiB 的固定輸送量容量。
- 持久性 SSD 儲存：建議用於需要最高水準可用性的任務關鍵型、長時間執行的工作負載，例如 HPC 模擬、大數據分析或Machine Learning訓練。使用持久性 SSD 儲存，您可以設定所需的儲存容量和輸送量容量 （每 TiB)。

效能考量：

- 管理 Pod 以管理 FSx for Lustre 檔案系統：設定已安裝 lustre 用戶端並掛載 FSx 檔案系統的「管理」Pod。這將使存取點能夠對 FSx 檔案系統進行微調，以及在您需要在啟動 GPU 運算執行個體之前，使用 ML 訓練資料或 LLM 模型預熱 FSx 檔案系統的情況下。如果您的架構使用 Spot 型 Amazon EC2 GPU/運算執行個體，其中您可以使用管理 Pod 將所需的資料「暖」或「預先載入」到 FSx 檔案系統，以便在執行 Spot 型 Amazon EC2 執行個體時準備好處理資料，這一點尤其重要。
- Elastic Fabric Adapter (EFA)：持久性 SSD 儲存體部署類型支援 Elastic Fabric Adapter (EFA)，其中使用 EFA 非常適合高效能和輸送量型 GPU 工作負載。請注意，FSx for Lustre 支援 NVIDIA

GPUDirect 儲存 (GDS)，其中，GDS 是一種技術，可在本機或遠端儲存體與 GPU 記憶體之間建立直接資料路徑，以加快資料存取速度。

- 壓縮：如果您有可以壓縮的檔案類型，請在檔案系統上啟用資料壓縮。這有助於提高效能，因為資料壓縮可減少 FSx for Lustre 檔案伺服器和儲存體之間傳輸的資料量。
- Lustre 檔案系統分割組態：
  - 資料分割：允許 FSx for Luster 將檔案的資料分散到 Lustre 檔案系統中的多個物件儲存目標 (OSTs)，將平行存取和輸送量最大化，尤其是針對大規模 ML 訓練任務。
  - 獨立檔案系統分割：根據預設，會透過 FSx for Lustre 的漸進式檔案配置 (PFL) 功能為您建立 4 元件 Lustre 分割組態。在大多數情況下，您不需要更新預設 PFL Lustre 條紋計數/大小。如果您需要調整 Lustre 資料分割，您可以參考 FSx for Lustre 檔案系統的分割參數，手動調整 Lustre 分割。
  - S3-Linked檔案系統：使用原生 Amazon S3 整合 （資料儲存庫關聯或 DRA) 匯入 FSx 檔案系統的檔案不使用預設 PFL 配置，而是在檔案系統的 `ImportedFileChunkSize` 參數中使用配置。大於 S3-imported檔案`ImportedFileChunkSize`將存放在多個 OSTs 上，並根據`ImportedFileChunkSize`定義的值 （預設 1GiB) 具有條紋計數。如果您有大型檔案，建議您將此參數調校為較高的值。
  - 配置：將 FSx for Lustre 檔案系統部署在與運算或 GPU 節點相同的可用區域中，以啟用資料的最低延遲存取，避免跨可用區域存取模式。如果您的多個 GPU 節點位於不同的可用區域，建議您在每個可用區域中部署 FSx 檔案系統，以實現低延遲的資料存取。

範例

FSx for Lustre 檔案系統的持久性磁碟區 (PV) 定義，使用靜態佈建 （其中已佈建 FSx 執行個體）。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: fsx-pv
spec:
  capacity:
    storage: 1200Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  mountOptions:
    - flock
  persistentVolumeReclaimPolicy: Recycle
  csi:
```

```
    driver: fsx.csi.aws.com
    volumeHandle: [FileSystemId of FSx instance]
    volumeAttributes:
      dnsname: [DNSName of FSx instance]
      mountname: [MountName of FSx instance]
```

範例

稱為 之 PV 的持久性磁碟區宣告定義fsx-pv：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: fsx-claim
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ""
  resources:
    requests:
      storage: 1200Gi
  volumeName: fsx-pv
```

範例

設定 Pod 以使用 的持久性磁碟區宣告fsx-claim：

```
apiVersion: v1
kind: Pod
metadata:
  name: fsx-app
spec:
  containers:
  - name: app
    image: amazonlinux:2023
    command: ["/bin/sh"]
    volumeMounts:
    - name: persistent-storage
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
```

```
      claimName: fsx-claim
```

如需完整範例，請參閱 GitHub 中的 FSx for Lustre 驅動程式範例。使用 Amazon CloudWatch 監控 Amazon FSx for Lustre 效能指標。 Amazon CloudWatch 發現效能瓶頸時，請視需要調整您的組態參數。

案例：單一 GPU 執行個體工作負載

搭配 CSI 驅動程式的 Amazon S3 掛載點：您可以使用適用於 Amazon S3 CSI 驅動程式的掛載點，將 Amazon S3 儲存貯體掛載為 Pod 中的磁碟區。此方法允許精細存取控制哪些 Pod 可以存取特定的 S3 儲存貯體。每個 Pod 都有自己的掛載點執行個體和本機快取 (5-10GB)，在 Pod 之間隔離模型載入和讀取效能。此設定支援使用 IAM Roles for Service Accounts (IRSA) 進行 Pod 層級身分驗證，並支援不同模型或客戶的獨立模型版本控制。權衡是記憶體用量和 API 流量的增加，因為每個 Pod 都會發出 S3 API 呼叫並維護自己的快取。

Pod 部署 YAML 搭配 CSI 驅動程式的部分範例：

```
# CSI driver dynamically mounts the S3 bucket for each pod

volumes:
  - name: s3-mount
    csi:
      driver: s3.csi.aws.com
      volumeAttributes:
        bucketName: your-s3-bucket-name
        mountOptions: "--allow-delete"  # Optional
        region: us-west-2

containers:
  - name: inference
    image: your-inference-image
    volumeMounts:
      - mountPath: /models
        name: s3-mount
volumeMounts:
  - name: model-cache
    mountPath: /models
volumes:
  - name: model-cache
    hostPath:
      path: /mnt/s3-model-cache
```

效能考量：

- 資料快取：S3 的掛載點可以快取內容，以降低成本並改善對相同檔案重複讀取的效能。如需快取選項和參數，請參閱快取組態。
- 物件部分大小：存放和存取大小超過 72GB 的檔案時，請參閱設定掛載點效能，了解如何設定 `--read-part-size`和 `--write-part-size`命令列參數以符合您的資料設定檔和工作負載需求。
- Shared-cache 專為大小上限為 1MB 的物件而設計。它不支援大型物件。使用本機快取選項在 NVMe 或 EKS 節點的 EBS 磁碟區中快取物件。
- API 請求費用：使用 S3 掛載點執行大量檔案操作時，API 請求費用可能會成為儲存成本的一部分。若要緩解這種情況，如果不需要強式一致性，請一律啟用中繼資料快取，並將 `metadata-ttl` API 操作次數設定為 S3。

如需詳細資訊，請參閱 Amazon EKS 官方文件中的 Amazon S3 CSI 驅動程式掛載點。如果發生瓶頸，我們建議您使用 Amazon CloudWatch 指標監控 Amazon S3 的效能指標 Amazon CloudWatch，並視需要調整您的組態。

Amazon FSx for OpenZFS 持久共用儲存

對於涉及多個 EC2 GPU 運算執行個體的案例，以及需要不同應用程式的高可用性、高效能、成本敏感度和多個 Pod 部署的延遲敏感工作負載，我們建議使用 Amazon FSx for OpenZFS。某些工作負載範例包括即時推論、強化學習和訓練生成對手網路。FSx for OpenZFS 特別適用於需要高效能存取聚焦目錄結構的工作負載，以及使用小型 IO 資料存取模式的小型檔案。此外，FSx for OpenZFS 提供獨立於儲存容量的彈性擴展效能，透過將儲存大小與實際需求配對，同時維持所需的效能等級，協助您達成最佳的成本效益

原生 FSx for OpenZFS CSI 驅動程式允許透過建立多個磁碟區，將多個 PVCs 建立到單一檔案系統。這可減少管理開銷，並透過單一檔案系統上的合併應用程式 Pod 部署，最大化檔案系統的輸送量和 IOPS 使用率。此外，它還包含企業功能，例如零複本快照、零複本複製，以及可透過 CSI 驅動程式動態佈建的使用者和群組配額。

FSx for OpenZFS 在建立時支援三種不同的部署類型：

- 單一可用區：具有低於毫秒延遲的最低成本選項，但檔案系統或可用區域層級不提供高可用性。建議用於開發和測試工作負載或在應用程式層具有高可用性的工作負載。
- 單一可用區 (HA)：以低於毫秒的延遲在檔案系統層級提供高可用性。建議用於需要高可用性的最高效能工作負載。

- 異地同步備份：在檔案系統層級以及跨可用區域提供高可用性。建議用於需要跨可用區域額外可用性的高效能工作負載。

效能考量：

- 部署類型：如果不需要跨可用區域的其他可用性，請考慮使用單一可用區域 (HA) 部署類型。此部署類型提供高達 100% 的寫入輸送量、維持低於毫秒的延遲，而且 Gen2 檔案系統具有額外的 NVMe 快取，可儲存高達 TB 的經常存取資料。多可用區域檔案系統提供高達 75% 的寫入輸送量，延遲增加，以容納跨可用區域流量。
- 輸送量和 IOPS：為檔案系統設定的輸送量和 IOPS 都可以在部署後向上或向下擴展。您可以佈建高達 10GB/s 的磁碟輸送量，提供高達 21GB/s 的快取資料存取。IOPS 可以從磁碟擴展到 400，000 個，快取可提供超過 100 萬個 IOPS。請注意，單一可用區檔案系統的輸送量擴展確實會導致檔案系統短暫中斷，因為沒有高可用性。單一可用區 (HA) 或多可用區檔案系統的輸送量擴展可以不中斷地完成。SSD IOPS 可以每六小時擴展一次。
- 儲存類別：FSx for OpenZFS 同時支援 SSD 儲存類別和 Intelligent-Tiering 儲存類別。對於 AI/ML 工作負載，建議您使用 SSD 儲存類別，為讓 CPU/GPU 保持忙碌的工作負載提供一致的效能。
- 壓縮：如果您的工作負載可以壓縮，請啟用 LZ4 壓縮演算法。這可減少每個檔案在快取中耗用的資料量，允許直接從快取提供更多資料做為網路輸送量，並降低 SSD 磁碟上的負載 IOPS。
- 記錄大小：大多數 AI/ML 工作負載將受益於保留預設的 128KiB 記錄大小。只有在資料集由大型檔案 (10GiB 以上） 組成，且應用程式的一致性小區塊存取低於 128KiB 時，才應降低此值。

建立檔案系統後，服務會自動建立相關聯的根磁碟區。最佳實務是將資料存放在檔案系統根磁碟區的子磁碟區中。使用 FSx for OpenZFS CSI 驅動程式，您可以建立相關聯的持久性磁碟區宣告，以動態建立子磁碟區。

範例：

FSx for OpenZFS 磁碟區的儲存類別 (SC) 定義，用於在現有檔案系統上建立根磁碟區的子磁碟區 ($ROOT_VOL_ID)，並使用 NFS v4.2 通訊協定將磁碟區匯出至 VPC CIDR ($VPC_CIDR)。

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fsxz-vol-sc
provisioner: fsx.openzfs.csi.aws.com
parameters:
  ResourceType: "volume"
  ParentVolumeId: '"$ROOT_VOL_ID"'
```

```
  CopyTagsToSnapshots: 'false'
  DataCompressionType: '"LZ4"'
  NfsExports: '[{"ClientConfigurations": [{"Clients": "$VPC_CIDR", "Options":
 ["rw","crossmnt","no_root_squash"]}]}]'
  ReadOnly: 'false'
  RecordSizeKiB: '128'
  Tags: '[{"Key": "Name", "Value": "AI-ML"}]'
  OptionsOnDeletion: '["DELETE_CHILD_VOLUMES_AND_SNAPSHOTS"]'
reclaimPolicy: Delete
allowVolumeExpansion: false
mountOptions:
  - nfsvers=4.2
  - rsize=1048576
  - wsize=1048576
  - timeo=600
  - nconnect=16
  - async
```

針對上述建立的 fsxz-vol-sc 動態建立持久性磁碟區宣告 (PVC)。請注意，配置的儲存容量為 1Gi，FSx for OpenZFS 磁碟區為必要項目，如 CSI 驅動程式常見問答集中所述。磁碟區將獲提供使用此組態佈建至檔案系統的完整容量。如果需要限制磁碟區容量，您可以使用使用者或群組配額來執行此操作。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic-vol-pvc
  namespace: example
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: fsxz-vol-sc
  resources:
    requests:
      storage: 1Gi
```

設定 Pod 以使用 dynamic-vol-pvc 的持久性磁碟區宣告 (PVC) 掛載磁碟區：

```
kind: Pod
apiVersion: v1
metadata:
  name: fsx-app
  namespace: example
spec:
```

```
  volumes:
    - name: dynamic-vol-pv
      persistentVolumeClaim:
        claimName: dynamic-vol-pvc
  containers:
    - name: app
      image: amazonlinux:2023
      command: ["/bin/sh"]
      volumeMounts:
        - mountPath: "/mnt/fsxz"
          name: dynamic-vol-pv
```

共用模型快取的 Amazon EFS

當您有多個 EC2 GPU 運算執行個體環境，且動態工作負載需要跨多個節點和可用區域共用模型存取（例如，使用 Karpenter 進行即時線上推論），且具有中等效能和可擴展性需求時，我們建議您透過 EFS CSI 驅動程式使用 Amazon Elastic File System (EFS) 檔案系統做為持久性磁碟區。Amazon EFS 是全受管、高可用性和可擴展的雲端型 NFS 檔案系統，可讓 EC2 執行個體和容器具有共用檔案儲存，並具有一致的效能，而且不需要預先佈建儲存體。使用 EFS 做為模型磁碟區，並透過在 EKS 叢集上定義持久性磁碟區，將磁碟區掛載為共用檔案系統。每個由 EFS 檔案系統支援的持久性磁碟區宣告 (PVC) 都會建立為 EFS 檔案系統的 EFS 存取點。EFS 允許多個節點和 Pod 存取相同的模型檔案，無需將資料同步到每個節點的檔案系統。安裝 EFS CSI 驅動程式，將 EFS 與 EKS 整合。

您可以使用下列輸送量模式部署 Amazon EFS 檔案系統：

- 爆量輸送量：擴展檔案系統大小的輸送量，適用於偶爾爆量的各種工作負載。
- 佈建輸送量：專用輸送量，非常適合具有可預測效能需求的一致 ML 訓練任務。
- 彈性輸送量（建議 ML 使用）：根據工作負載、不同 ML 工作負載的成本效益自動擴展。

若要檢視效能規格，請參閱 Amazon EFS 效能規格。

效能考量：

- 將彈性輸送量用於不同的工作負載。
- 針對作用中 ML 工作負載使用標準儲存類別。

如需在 EKS 叢集和 Pod 中使用 Amazon EFS 檔案系統作為持久性磁碟區的完整範例，請參閱 GitHub 中的 EFS CSI 驅動程式範例。使用 Amazon CloudWatch 監控 Amazon EFS 效能指標。 Amazon CloudWatch 發現效能瓶頸時，請視需要調整您的組態參數。

將 S3 Express One Zone 用於延遲敏感的物件導向工作流程

對於 Amazon EKS 上的延遲敏感 AI/ML 工作負載，例如大規模模型訓練、推論或高效能分析，建議使用 S3 Express One Zone 進行高效能模型儲存和擷取。S3 Express One Zone 提供階層式命名空間，例如檔案系統，您只需上傳到目錄儲存貯體，適合「卡入所有內容」，同時維持高速。如果您熟悉物件導向工作流程，此功能特別有用。或者，如果您更習慣檔案系統（例如 POSIX 相容），您可能偏好 Amazon FSx for Lustre 或 OpenZFS。Amazon S3 Express One Zone 使用目錄儲存貯體將資料存放在單一可用區域 (AZ)，並提供比標準 S3 儲存貯體更低的延遲，該儲存貯體會將資料分散到多個AZs。為了獲得最佳結果，請務必將 EKS 運算與 Express One Zone 儲存貯體位於相同的可用區域中。若要進一步了解 S3 Express One Zone 的差異，請參閱目錄儲存貯體的差異。

若要使用檔案系統語意存取 S3 Express One Zone，建議使用掛載點 S3 CSI 驅動程式，該驅動程式會將 S3 儲存貯體（包括 Express One Zone) 掛載為本機檔案系統。這會將檔案操作（例如，開啟、讀取、寫入）轉換為 S3 API 呼叫，為多個用戶端的大量讀取工作負載和循序寫入新物件提供最佳化的高輸送量存取。如需支援的操作和限制的詳細資訊（例如，沒有完整的 POSIX 合規，但附加和重新命名 Express One Zone 中支援），請參閱掛載點語意文件。

效能優勢

- 提供比 S3 Standard 快 10 倍的資料存取，具有一致的單一位數毫秒延遲，以及高達 80% 的請求成本。
- 擴展以處理每個目錄儲存貯體每秒數十萬到數百萬個請求，避免在極端載入期間在標準 S3 中看到限流或中斷（例如，從具有數十到數十萬GPUs/CPUs 飽和網路的叢集）。
- 使用工作階段型身分驗證機制。驗證一次以取得工作階段字符，然後以高速執行重複操作，而不會產生每次請求的身分驗證額外負荷。這針對頻繁檢查點或資料載入等工作負載進行了最佳化。

建議的使用案例

- 快取：搭配 S3 Express One Zone 使用掛載點 S3 CSI 驅動程式的其中一個常用案例是快取。第一個執行個體會從 S3 Standard（一般用途）讀取資料，並在低延遲的 Express One Zone 中快取資料。其他用戶端後續讀取會更快地存取快取的資料，這非常適合多個 EKS 節點讀取相同資料（例如共用訓練資料集）的多節點案例。對於重複存取和降低運算成本，這可以提高高達 7 倍的效能。對於需要完全 POSIX 合規的工作負載（例如檔案鎖定和就地修改），請將 Amazon FSx for Lustre 或 OpenZFS 視為替代方案。
- 大規模 AI/ML 訓練和推論：非常適合具有數百個或數千個運算節點（例如 EKS 叢集中的 GPUs) 的工作負載，其中一般用途 S3 限流可能會導致延遲，浪費昂貴的運算資源。例如，執行每日模型測

試/檢查點的 LLM 研究人員或組織受益於快速、可靠的存取，而不會破壞區域 S3。對於較小的工作負載（例如 10 個節點），S3 Standard 或其他儲存類別可能已足夠。

- 資料管道：載入/準備模型、封存訓練資料或串流檢查點。如果您的團隊偏好物件儲存，而非傳統檔案系統（例如，由於熟悉 S3)，請針對 FSx for Lustre 等 POSIX 相容選項，使用此選項而非工程變更。

考量

- 彈性：單一可用區設計提供 99.999999999% 的耐用性（與標準 S3 相同，透過 AZ 中的備援），但相較於多可用區類別 (99.99% 的可用性），可用性較低 (99.95% 的設計、99.9% 的 SLA)。它對 AZ 故障的彈性較低。用於可重新建立或快取的資料。考慮對關鍵工作負載進行多可用區複寫或備份。
- API 和功能支援：支援 S3 API 的子集（例如，沒有生命週期政策或複寫）；可能需要對工作階段身分驗證或物件處理進行次要應用程式變更。 APIs
- EKS 整合：將 EKS Pod/節點與目錄儲存貯體位於相同的可用區域中，以將網路延遲降至最低。使用適用於 Amazon S3 或 CSI 驅動程式的掛載點進行 Kubernetes 原生存取。
- 測試：在非生產環境中測試擷取延遲，以驗證效能提升。在標準 S3 案例中監控限流（例如高 GPU 飽和度）並比較。

S3 Express One Zone 儲存類別可在多個區域中使用，並針對需要物件存取的工作負載與 EKS 整合，而無需等待儲存。若要進一步了解，請參閱 S3 Express One Zone 入門。

可觀測性

> **Tip**
>
> 註冊即將舉行的 Amazon EKS AI/ML 研討會。

監控與可觀測性

GPU 指標說明

GPU 使用率指標會顯示 GPU 在範例時段期間是否執行任何工作。此指標會擷取 GPU 執行至少一個指令的時間百分比，但不會顯示 GPU 使用其硬體的效率。GPU 包含多個串流多處理器 (SMs)，這是執行指示的平行處理單元。100% 使用率讀數可能表示 GPU 在其所有 SMs 中執行繁重的平行工作負

載，也可能表示在範例期間內啟動 GPU 的單一小型指令。若要了解實際使用率，您需要在硬體架構的多個層級檢查 GPU 指標。每個串流多處理器都是由不同的核心類型所建置，而且每個 layer 都會公開不同的效能特性。最上層指標 (GPU 使用率、記憶體使用率、GPU Power 和 GPU 溫度，透過 nvidia-smi 可見）會顯示裝置是否處於作用中狀態。更深入的指標 (SM 使用率、SM 活動和張量核心使用量）會顯示 GPU 使用其資源的效率。

以高 GPU 電源用量為目標

未充分利用的 GPUs 會浪費運算容量並增加成本，因為工作負載無法同時參與所有 GPU 元件。對於 Amazon EKS 上的 AI/ML 工作負載，請將 GPU 用電量追蹤為代理，以識別實際的 GPU 活動。GPU 使用率會報告 GPU 執行任何核心的時間百分比，但不會顯示串流多處理器、記憶體控制器和張量核心是否同時處於作用中狀態。用電量會公開此差距，因為完全參與的硬體比執行輕量型核心或任務之間閒置的硬體消耗更多電力。比較耗電量與 GPU 的熱設計能力 (TDP)，以發現使用率不足，然後調查您的工作負載是否因 CPU 預先處理、網路 I/O 或效率不佳的批次大小而遇到瓶頸。

在 Amazon EKS 上設定 CloudWatch Container Insights，以識別 GPU 耗電量低的 Pod、節點或工作負載。此工具直接與 Amazon EKS 整合，可讓您監控 GPU 耗電量，並在耗電量低於目標層級時調整 Pod 排程或執行個體類型。如果您需要進階視覺化或自訂儀表板，請使用 NVIDIA 的 DCGM-Exporter 搭配 Prometheus 和 Grafana 進行 Kubernetes 原生監控。兩者都接近表面金鑰 NVIDIA 指標，例如 `nvidia_smi_power_draw`(GPU 耗電量）和 `nvidia_smi_temperature_gpu`(GPU 溫度）。如需指標清單，請瀏覽 https：//https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-Agent-NVIDIA-GPU.htm。尋找模式，例如在特定時間或特定任務中持續保持低功耗。這些趨勢可協助您識別合併工作負載或調整資源配置的位置。

Kubernetes 中的靜態資源限制（例如 CPU、記憶體和 GPU 計數）通常會導致過度佈建或利用率不足，尤其是在需求波動的推論等動態 AI/ML 工作負載。分析您的使用率趨勢，並將工作負載合併到較少的 GPUs。在配置其他 GPU 之前，請確定每個 GPU 都達到完整的使用率。此方法可減少浪費並降低成本。如需最佳化排程和共用策略的詳細指引，請參閱 EKS Compute and Autoscaling 最佳實務

可觀測性和指標

使用 AI/ML 工作負載的監控和可觀測性工具

現代 AI/ML 服務需要跨基礎設施、建模和應用程式邏輯進行協調。平台工程師管理基礎設施和可觀測性堆疊。它們收集、存放和視覺化指標。AI/ML 工程師定義模型特定的指標，並在不同的負載和資料分佈下監控效能。應用程式開發人員會使用 APIs、路由請求，以及追蹤服務層級指標和使用者互動。如果沒有統一的可觀測性實務，這些團隊會在孤島中工作，並錯過有關系統運作狀態和效能的關鍵訊號。建立跨環境的共用可見性，可確保所有利益相關者都能及早偵測問題，並維持可靠的服務。

最佳化 AI/ML 工作負載的 Amazon EKS 叢集帶來獨特的監控挑戰，尤其是 GPU 記憶體管理。如果沒有適當的監控，組織會面臨out-of-memory(OOM) 錯誤、資源效率低下和不必要的成本。有效的監控可確保 EKS 客戶有更好的效能、彈性和更低的成本。使用結合三個監控層的整體方法。首先，使用 NVIDIA DCGM Exporter 監控精細的 GPU 指標，以追蹤 GPU 用電量、GPU 溫度、SM 活動、SM 佔用率和 XID 錯誤。其次，監控 Ray 和 vLLM 等推論服務架構，透過其原生指標取得分散式工作負載洞見。第三，收集應用程式層級的洞見，以追蹤工作負載特定的自訂指標。這種分層方法可讓您從硬體使用率到應用程式效能的可見性。

工具和架構

數個工具和架構提供原生out-of-the-box指標，用於監控 AI/ML 工作負載。這些內建指標可免除自訂檢測的需求，並縮短設定時間。這些指標著重於效能層面，例如延遲、輸送量和權杖產生，這對於推論服務和基準測試至關重要。使用原生指標可讓您立即開始監控，而無需建置自訂集合管道。

- vLLM：適用於大型語言模型 (LLMs) 的高輸送量服務引擎，可提供請求延遲和記憶體用量等原生指標。
- Ray：分散式運算架構，可發出可擴展 AI 工作負載的指標，包括任務執行時間和資源使用率。
- Hugging Face Text Generation Inference (TGI)：用於部署和服務 LLMs 的工具組，具有內建的推論效能指標。
- NVIDIA genai-perf：一種命令列工具，用於基準化生成式 AI 模型、測量輸送量、延遲和 LLM 特定指標，例如在特定時間間隔內完成的請求。

可觀測性方法

我們建議以下列其中一種方式實作任何其他可觀測性機制。

CloudWatch Container Insights 如果您的組織偏好設定最少的 AWS 原生工具，我們建議您使用 CloudWatch Container Insights。它與 NVIDIA DCGM Exporter 整合，以收集 GPU 指標，並提供預先建置的儀表板以快速洞察。透過在叢集上安裝 CloudWatch Observability 附加元件來啟用，Container Insights 會部署和管理 NVIDIA DCGM Exporter 的生命週期，從 Nvidia 的驅動程式收集 GPU 指標並將其公開給 CloudWatch。

安裝 Container Insights 之後，CloudWatch 會自動偵測您環境中的 NVIDIA GPUs，並收集重要的運作狀態和效能指標。這些指標會顯示在已整理out-of-the-box儀表板上。您也可以使用統一 CloudWatch Agent 將 Ray 和 vLLM 與 CloudWatch 整合，以傳送其原生指標。 CloudWatch 這種統一的方法簡化了 EKS 環境中的可觀測性，並讓團隊專注於效能調校和成本最佳化，而不是建置監控基礎設施。

如需可用指標的完整清單，請參閱 Amazon EKS 和 Kubernetes Container Insights 指標。如需實作 GPU step-by-step指引，請參閱使用 Amazon CloudWatch Container Insights 取得 NVIDIA GPU 工作

負載的操作洞見。如需最佳化推論延遲的實際範例，請參閱最佳化 AI 回應能力：Amazon Bedrock 延遲最佳化推論的實際指南。

受管 Prometheus 和 Grafana 如果您的組織需要自訂儀表板和進階視覺化功能，請使用 NVIDIA DCGM-Exporter 和 Grafana 部署 Prometheus 以進行 Kubernetes 原生監控。Prometheus 會從 DCGM-Exporter 抓取和存放 GPU 指標，而 Grafana 則提供靈活的視覺化和提醒功能。相較於 CloudWatch Container Insights，此方法可讓您更能控制儀表板設計和指標保留。

您可以整合 Ray、vLLM Ray 和 vLLM 等開放原始碼架構，將其原生指標匯出至 Prometheus，以擴展此監控堆疊。您也可以將 Grafana 連線至 AWS X-Ray 資料來源，以視覺化分散式追蹤，並識別推論管道的效能瓶頸。此組合透過應用程式層級請求流程，提供 GPU 層級指標end-to-end可見性。

如需部署此監控堆疊的step-by-step指引，請參閱使用 AWS 受管開放原始碼服務在 Amazon EKS 上監控 GPU 工作負載。

考慮監控核心訓練和微調指標

監控核心訓練指標，以追蹤 Amazon EKS 叢集的運作狀態和效能，以及其上執行的機器學習工作負載。訓練工作負載與推論工作負載具有不同的監控需求，因為它們會長時間執行、以不同的方式使用資源，而且需要模型收斂和資料管道效率的可見性。以下指標可協助您識別瓶頸、最佳化資源配置，並確保訓練任務成功完成。如需實作此監控方法step-by-step指引，請參閱在 Amazon EKS 上觀察機器學習工作負載簡介。

資源用量指標

監控資源用量指標，以驗證您的資源是否已正確使用。這些指標可協助您識別瓶頸和根本原因效能問題。

- CPU、記憶體、網路、GPU Power 和 GPU 溫度 - 監控這些指標，以確保配置的資源符合工作負載需求，並識別最佳化機會。追蹤 gpu_memory_usage_bytes 等指標，以識別記憶體消耗模式並偵測尖峰用量。計算百分位數，例如第 95 個百分位數 (P95)，以了解訓練期間最高的記憶體需求。此分析可協助您最佳化模型和基礎設施，以避免 OOM 錯誤並降低成本。
- SM 佔用、SM 活動、FPxx 活動 - 監控這些指標，以了解 GPU 上的基礎資源的使用方式。SM 活動的目標 0.8 作為經驗法則。
- 節點和 Pod 資源使用率 - 追蹤節點和 Pod 層級的資源使用量，以識別資源爭用和潛在的瓶頸。監控節點是否接近容量限制，這可能會延遲 Pod 排程和緩慢的訓練任務。
- 資源使用率與請求和限制比較 — 將實際資源使用量與設定的請求和限制進行比較，以確定您的叢集是否可以處理目前的工作負載並容納未來的工作負載。此比較顯示是否需要調整資源配置，以避免 OOM 錯誤或資源浪費。

- ML 架構的內部指標 - 從 TensorFlow 和 PyTorch 等 ML 架構擷取內部訓練和收斂指標。這些指標包括損失曲線、學習率、批次處理時間和訓練步驟持續時間。使用 TensorBoard 或類似工具視覺化這些指標，以追蹤模型收斂並識別訓練效率低下。

模型效能指標

監控模型效能指標，以驗證您的訓練程序是否產生符合準確性和業務需求的模型。這些指標可協助您判斷何時停止訓練、比較模型版本，以及識別效能降低。

- 準確性、精確度、召回和 F1-score — 追蹤這些指標，以了解模型對驗證資料的效能。在每個訓練 epoch 之後計算驗證集上的 F1-score，以評估模型是否正在改善，以及何時達到可接受的效能等級。
- 業務特定指標和 KPIs — 定義和追蹤直接測量 AI/ML 計畫商業價值的指標。對於建議系統，追蹤點擊率或轉換率等指標，以確保模型推動預期的業務成果。
- 隨時間推移的效能 — 比較跨模型版本和訓練執行的效能指標，以識別趨勢並偵測降級。相較於基準模型，追蹤較新的模型版本是否維持或改善效能。此歷史比較可協助您決定是否部署新模型或調查訓練問題。

資料品質和偏離指標

監控資料品質和偏離指標，以確保您的訓練資料保持一致且具代表性。資料偏離可能會導致模型效能隨著時間降低，而資料品質問題可能會阻止模型收斂或產生不可靠的結果。

- 輸入資料的統計屬性 — 追蹤一段時間內輸入特徵的平均值、標準差和分佈等統計屬性，以偵測資料偏離或異常。監控特徵分佈是否從您的基準訓練資料大幅轉移。例如，如果關鍵特徵的平均值變更超過兩個標準差，請調查您的資料管道是否已變更，或基礎資料來源是否已轉移。
- 資料偏離偵測和提醒 — 實作自動化機制，在資料品質問題影響訓練之前對其進行偵測和提醒。使用統計測試，例如 Kolmogorov-Smirnov 測試或卡方測試，將目前的資料分佈與原始訓練資料進行比較。在測試偵測到重大偏離時設定提醒，以便您可以使用更新的資料重新訓練模型或調查資料管道問題。

延遲和輸送量指標

監控延遲和輸送量指標，以識別訓練管道中的瓶頸，並最佳化資源使用率。這些指標可協助您了解在訓練期間花費的時間，以及專注於最佳化工作的位置。

- ML 訓練管道的End-to-End延遲 — 測量資料流經整個訓練管道的總時間，從資料擷取到模型更新。跨訓練執行追蹤此指標，以識別管道變更是否改善或降低效能。高延遲通常表示節點之間的資料載入、預先處理或網路通訊存在瓶頸。
- 訓練輸送量和處理率 — 追蹤訓練管道每單位時間處理的資料量，以確保有效率的資源使用率。監控指標，例如每秒處理的樣本或每分鐘完成的批次。相對於硬體容量而言，低輸送量表示資料載入、預先處理或模型運算會浪費 GPU 週期的效率低下。
- 檢查點儲存和還原延遲 – 監控將模型檢查點儲存到儲存體 (S3、EFS、FSx) 所需的時間，並在恢復任務或從故障中復原時將其還原到 GPU 或 CPU 記憶體。緩慢的檢查點操作會延長任務復原時間並增加成本。追蹤檢查點大小、節省持續時間、還原持續時間和失敗計數，以識別最佳化機會，例如壓縮或更快的儲存層。
- 資料載入和預先處理時間 - 測量從儲存體載入資料和套用預先處理轉換所花費的時間。將此時間與模型運算時間進行比較，以判斷您的訓練是資料限制還是運算限制。如果資料載入耗用超過總訓練時間的 20%，請考慮使用快取、預先擷取或更快的儲存來最佳化資料管道。

錯誤率和失敗

監控整個訓練管道的錯誤率和故障，以維持可靠性並防止浪費的運算資源。未偵測到的錯誤可能會導致訓練任務無提示地失敗、產生無效的模型，或在發現問題之前浪費數小時的 GPU 時間。

- 管道錯誤監控 — 追蹤 ML 管道所有階段的錯誤，包括資料預先處理、模型訓練和檢查點操作。記錄錯誤類型、頻率和受影響的元件，以快速識別問題。常見的錯誤包括資料格式不符、預先處理期間 out-of-memory故障，以及檢查點儲存因儲存限制而失敗。設定錯誤率超過基準閾值時的提醒，以便在錯誤層疊之前進行調查。
- 重複錯誤分析 — 識別和調查重複錯誤中的模式，以防止未來失敗並改善管道可靠性。分析日誌，以找出特定資料範例、批次大小或訓練組態是否一致地導致失敗。例如，如果某些輸入資料類型觸發預先處理錯誤，請在管道中稍早新增驗證檢查，或更新您的資料清理邏輯。追蹤故障之間的平均時間 (MTBF)，以測量管道可靠性是否隨著時間改善。」

Kubernetes 和 EKS 特定指標

監控 Kubernetes 和 EKS 指標，以確保您的叢集基礎設施正常運作，並可支援您的訓練工作負載。這些指標可協助您在基礎設施問題導致訓練任務失敗或效能降低之前對其進行偵測。

- Kubernetes 叢集狀態指標 — 監控 Kubernetes 物件的運作狀態和狀態，包括 Pod、節點、部署和服務。追蹤 Pod 狀態以識別停滯在待定、失敗或當機迴圈狀態的 Pod。監控節點條件，以偵測磁碟壓力、記憶體壓力或網路無法使用等問題。使用 kubectl 或監控工具持續檢查這些指標，並在 Pod 無法啟動或節點變得無法排程時設定提醒。

- 訓練管道執行指標 — 追蹤成功和失敗的管道執行、任務持續時間、步驟完成時間和協同運作錯誤。監控訓練任務是否在預期時段內完成，以及失敗率是否隨時間增加。追蹤指標，例如任務成功率、平均任務持續時間和失敗時間。這些指標可協助您識別基礎設施問題、組態問題或資料品質問題是否會導致訓練失敗。
- AWS 服務指標 — 追蹤支援 EKS 基礎設施和訓練工作負載之 AWS 服務的指標。監控 S3 指標，例如請求延遲、錯誤率和輸送量，以確保資料載入效能保持一致。追蹤 EBS 磁碟區指標，包括 IOPS、輸送量和佇列長度，以偵測儲存瓶頸。監控 VPC 流量日誌和網路指標，以識別節點或外部服務之間的連線問題。
- Kubernetes 控制平面指標 — 監控 API 伺服器、排程器、控制器管理員等資料庫，以偵測影響叢集操作的效能問題或故障。追蹤 API 伺服器請求延遲、請求率和錯誤率，以確保控制平面快速回應排程請求。監控刻印資料庫大小、遞交持續時間和領導者變更，以偵測穩定性問題。高 API 伺服器延遲或頻繁的等推領導者變更可能會延遲 Pod 排程並延長訓練任務啟動時間。

應用程式和執行個體日誌

收集和分析應用程式和執行個體日誌，以診斷指標本身無法解釋的問題。日誌提供有關錯誤、狀態變更和系統事件的詳細內容，協助您了解訓練任務失敗或效能不佳的原因。將日誌與指標建立關聯可讓您更快地找出根本原因。

- 應用程式日誌 - 從訓練任務、資料管道和 ML 架構收集應用程式日誌，以識別瓶頸並診斷故障。這些日誌會擷取任務執行的詳細資訊，包括資料載入錯誤、模型初始化失敗、檢查點儲存錯誤，以及架構特定的警告。將日誌時間戳記與指標峰值建立關聯，以了解造成效能降低或故障的原因。例如，如果 GPU 使用率突然下降，請檢查應用程式日誌是否有錯誤，指出資料管道停滯或預先處理失敗。使用 CloudWatch Logs 或 Fluent Bit 等集中式記錄工具來彙總所有 Pod 的日誌，並使其可供搜尋。
- 執行個體日誌 - 收集執行個體層級日誌，例如系統日誌日誌和 dmesg 輸出，以偵測硬體問題和核心層級問題。這些日誌會顯示 GPU 驅動程式錯誤、記憶體配置失敗、磁碟 I/O 錯誤，以及應用程式日誌中可能不會出現的網路界面問題等問題。將執行個體日誌與應用程式日誌和指標建立關聯，以判斷訓練失敗是否來自硬體問題或應用程式問題。例如，如果訓練任務因out-of-memory錯誤而失敗，請檢查核心 OOM 刪除器訊息的 dmesg 日誌，指出系統是否記憶體不足，或應用程式是否超過其容器限制。設定重大硬體錯誤的提醒，例如 GPU XID 錯誤或磁碟故障，以便在造成廣泛的訓練中斷之前取代失敗的執行個體。

下列各節說明如何使用兩種 AWS 建議的方法收集上述指標：CloudWatch Container Insights 和 Amazon Managed Prometheus 與 Amazon Managed Grafana。 https://docs.aws.amazon.com/grafana/latest/userguide/getting-started-with-AMG.html如果您偏好具有最少設定和預先建置儀表板的 AWS 原生工具，請選擇 CloudWatch Container Insights。如果您需要自訂儀表板、進階視覺化功

能，或想要與現有的 Prometheus 型監控基礎設施整合，請選擇 Amazon Managed Prometheus with Amazon Managed Grafana。如需可用 Container Insights 指標的完整清單，請參閱 Amazon EKS 和 Kubernetes Container Insights 指標。

考慮監控即時線上推論指標

在即時系統中，低延遲對於及時回應使用者或其他相依系統至關重要。高延遲可能會降低使用者體驗或違反效能要求。影響推論延遲的元件包括模型載入時間、預先處理時間、實際預測時間、後製處理時間、網路傳輸時間。我們建議監控推論延遲，以確保符合服務層級協議 (SLAs) 的低延遲回應，並為下列項目開發自訂指標。在預期負載下進行測試，包括網路延遲、考慮並行請求，以及使用不同的批次大小進行測試。

- 首次權杖的時間 (TTFT) — 從使用者提交請求到收到回應開始的時間長度（第一個單字、權杖或區塊）。例如，在聊天機器人中，您會檢查使用者提出問題後產生第一段輸出（權杖）所需的時間。
- End-to-End延遲：這是從收到請求到傳送回回應的總時間。例如，測量從請求到回應的時間。
- 每秒輸出權杖 (TPS) — 指出模型開始回應後產生新權杖的速度。例如，在聊天機器人中，您可以追蹤基準文字的語言模型產生速度。
- 錯誤率 — 追蹤失敗的請求，這可能表示效能問題。例如，監控大型文件或特定字元的失敗請求。
- 輸送量 — 測量系統每單位時間可處理的請求或操作數量。例如，追蹤每秒請求以處理尖峰負載。

K/V（金鑰/值）快取可以是強大的推論延遲最佳化技術，尤其與轉換器型模型相關。K/V 快取會存放先前轉換器層運算的金鑰和值張量，減少自動迴歸推論期間的備援運算，尤其是大型語言模型 LLMs)。快取效率指標（特別是 K/V 或工作階段快取使用）：

- 快取命中/遺失率 — 對於利用快取 (K/V 或內嵌快取）的推論設定，測量快取的協助頻率。低命中率可能表示快取組態或工作負載變更不佳，這兩者都可能增加延遲。

在後續主題中，我們會示範如何收集上述幾個指標的資料。我們將提供兩種 AWS 建議方法的範例：AWS 原生 CloudWatch Container Insights 和開放原始碼 Amazon Managed Prometheus with Amazon Managed Grafana。您可以根據整體可觀測性需求選擇其中一個解決方案。如需 Container Insights 指標的完整清單，請參閱 Amazon EKS 和 Kubernetes Container Insights 指標。

追蹤 GPU 記憶體用量

如the section called “考慮監控核心訓練和微調指標”主題所述，GPU 記憶體使用量對於防止out-of-memory(OOM) 錯誤並確保有效率的資源使用率至關重要。下列範例示範如何檢測您的訓練應用程式，

以公開自訂長條圖指標 gpu_memory_usage_bytes，並計算 P95 記憶體用量以識別尖峰耗用量。請務必在預備環境中使用範例訓練任務（例如微調轉換器模型）進行測試。

AWS 原生 CloudWatch Container Insights 範例

此範例示範如何使用 AWS 原生方法檢測您的訓練應用程式，以長條圖gpu_memory_usage_bytes的形式公開。請注意，您的 AI/ML 容器必須設定為以 CloudWatch Embedded Metrics Format (EMF) 格式發出結構化日誌。CloudWatch 日誌會剖析 EMF 並發佈指標。在訓練應用程式中使用 aws_embedded_metrics，將 EMF 格式的結構化日誌傳送至擷取 GPU 指標的 CloudWatch Logs。

```
from aws_embedded_metrics import metric_scope
import torch
import numpy as np

memory_usage = []

@metric_scope
def log_gpu_memory(metrics):
    # Record current GPU memory usage
    mem = torch.cuda.memory_allocated()
    memory_usage.append(mem)

    # Log as histogram metric
    metrics.set_namespace("MLTraining/GPUMemory")
    metrics.put_metric("gpu_memory_usage_bytes", mem, "Bytes", "Histogram")

    # Calculate and log P95 if we have enough data points
    if len(memory_usage) >= 10:
        p95 = np.percentile(memory_usage, 95)
        metrics.put_metric("gpu_memory_p95_bytes", p95, "Bytes")
        print(f"Current memory: {mem} bytes, P95: {p95} bytes")

# Example usage in training loop
for epoch in range(20):
    # Your model training code would go here
    log_gpu_memory()
```

Prometheus 和 Grafana 範例

此範例示範如何使用 Python 中的 Prometheus 用戶端程式庫，檢測您的訓練應用程式以長條圖gpu_memory_usage_bytes`形式公開。

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import pynvml

start_http_server(8080)
memory_usage = Histogram(
    'gpu_memory_usage_bytes',
    'GPU memory usage during training',
    ['gpu_index'],
    buckets=[1e9, 2e9, 4e9, 8e9, 16e9, 32e9]
)

# Function to get GPU memory usage
def get_gpu_memory_usage():
    if torch.cuda.is_available():
        # Get the current GPU device
        device = torch.cuda.current_device()

        # Get memory usage in bytes
        memory_allocated = torch.cuda.memory_allocated(device)
        memory_reserved = torch.cuda.memory_reserved(device)

        # Total memory usage (allocated + reserved)
        total_memory = memory_allocated + memory_reserved

        return device, total_memory
    else:
        return None, 0

# Get GPU memory usage
gpu_index, memory_used = get_gpu_memory_usage()
```

追蹤即時線上推論的推論請求持續時間

如the section called “考慮監控核心訓練和微調指標”主題所述，低延遲對於為使用者或其他相依系統提供及時回應至關重要。下列範例示範如何追蹤由推論應用程式`inference_request_duration_seconds`公開的自訂長條圖指標。計算第 95 個百分位數 (P95) 延遲以專注於最壞情況，在預備環境中使用合成推論請求（例如，透過 Locust) 進行測試，並設定警示閾值（例如，>500 毫秒）以偵測 SLA 違規。

AWS 原生 CloudWatch Container Insights 範例

此範例示範如何使用 AWS CloudWatch Embedded Metric Format 在 inference_request_duration_seconds 的推論應用程式中建立自訂長條圖指標。

```
import boto3
import time
from aws_embedded_metrics import metric_scope, MetricsLogger

cloudwatch = boto3.client('cloudwatch')

@metric_scope
def log_inference_duration(metrics: MetricsLogger, duration: float):
    metrics.set_namespace("ML/Inference")
    metrics.put_metric("inference_request_duration_seconds", duration, "Seconds",
 "Histogram")
    metrics.set_property("Buckets", [0.1, 0.5, 1, 2, 5])

@metric_scope
def process_inference_request(metrics: MetricsLogger):
    start_time = time.time()

    # Your inference processing code here
    # For example:
    # result = model.predict(input_data)

    duration = time.time() - start_time
    log_inference_duration(metrics, duration)

    print(f"Inference request processed in {duration} seconds")

# Example usage
process_inference_request()
```

Prometheus 和 Grafana 範例

此範例示範如何使用 Python 中的 Prometheus 用戶端程式庫，在推論應用程式中為 inference_request_duration_seconds 建立自訂長條圖指標：

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import time

start_http_server(8080)
request_duration = Histogram(
```

```
    'inference_request_duration_seconds',
    'Inference request latency',
    buckets=[0.1, 0.5, 1, 2, 5]
)
start_time = time.time()
# Process inference request
request_duration.observe(time.time() - start_time)
```

在 Grafana 中，使用查詢`histogram_quantile(0.95, sum(rate(inference_request_duration_seconds_bucket[5m])) by (le, pod))`將 P95 延遲趨勢視覺化。若要進一步了解，請參閱 Prometheus 直方圖文件和 Prometheus 用戶端文件。

追蹤即時線上推論的字符輸送量

如the section called “考慮監控核心訓練和微調指標”主題所述，我們建議監控字符處理時間，以衡量模型效能並最佳化擴展決策。下列範例示範如何追蹤您的推論應用程式`token_processing_duration_seconds`公開的自訂長條圖指標 。計算第 95 個百分位數 (P95) 持續時間以分析處理效率、在非生產叢集中使用模擬請求負載（例如每秒 100 到 1000 個請求）進行測試，以及調整 KEDA 觸發以最佳化擴展。

AWS 原生 CloudWatch Container Insights 範例

此範例示範如何使用 AWS CloudWatch Embedded Metric Format 在 token_processing_duration_seconds 的推論應用程式中建立自訂長條圖指標。它使用維度 (`set_dimension) with a custom `get_duration_bucket 函數將持續時間分類為儲存貯體（例如 "⇐0.01"、">1")。

```
import boto3
import time
from aws_embedded_metrics import metric_scope, MetricsLogger

cloudwatch = boto3.client('cloudwatch')

@metric_scope
def log_token_processing(metrics: MetricsLogger, duration: float, token_count: int):
    metrics.set_namespace("ML/TokenProcessing")
    metrics.put_metric("token_processing_duration_seconds", duration, "Seconds")
    metrics.set_dimension("ProcessingBucket", get_duration_bucket(duration))
    metrics.set_property("TokenCount", token_count)

def get_duration_bucket(duration):
    buckets = [0.01, 0.05, 0.1, 0.5, 1]
```

```
    for bucket in buckets:
        if duration <= bucket:
            return f"<={bucket}"
    return f">{buckets[-1]}"

@metric_scope
def process_tokens(input_text: str, model, tokenizer, metrics: MetricsLogger):
    tokens = tokenizer.encode(input_text)
    token_count = len(tokens)

    start_time = time.time()
    # Process tokens (replace with your actual processing logic)
    output = model(tokens)
    duration = time.time() - start_time

    log_token_processing(metrics, duration, token_count)
    print(f"Processed {token_count} tokens in {duration} seconds")
    return output
```

Prometheus 和 Grafana 範例

此範例示範如何在推論應用程式中使用 Python 中的 Prometheus 用戶端程式庫為 token_processing_duration_seconds 建立自訂長條圖指標。

```
from prometheus_client import Histogram
from prometheus_client import start_http_server
import time

start_http_server(8080)
token_duration = Histogram(
    'token_processing_duration_seconds',
    'Token processing time per request',
    buckets=[0.01, 0.05, 0.1, 0.5, 1]
)
start_time = time.time()
# Process tokens
token_duration.observe(time.time() - start_time)
```

在 Grafana 中，使用查詢`histogram_quantile(0.95, sum(rate(token_processing_duration_seconds_bucket[5m])) by (le, pod))`將 P95 處理時間趨勢視覺化。若要進一步了解，請參閱 Prometheus 直方圖文件和 Prometheus 用戶端文件。

測量檢查點還原延遲

如the section called “考慮監控核心訓練和微調指標”主題所述，檢查點延遲是模型生命週期的多個階段期間的關鍵指標。下列範例示範如何追蹤應用程式`checkpoint_restore_duration_seconds`公開的自訂長條圖指標 。計算第 95 個百分位數 (P95) 持續時間以監控還原效能、在非生產叢集中使用 Spot 中斷進行測試，並設定警示閾值 （ 例如 <30 秒 ） 以偵測延遲。

AWS 原生 CloudWatch Container Insights 範例

此範例示範如何使用 CloudWatch Insights 檢測批次應用程式，以長條圖的形式公開 checkpoint_restore_duration_seconds：

```
import boto3
import time
import torch
from aws_embedded_metrics import metric_scope, MetricsLogger

@metric_scope
def log_checkpoint_restore(metrics: MetricsLogger, duration: float):
    metrics.set_namespace("ML/ModelOperations")
    metrics.put_metric("checkpoint_restore_duration_seconds", duration, "Seconds",
 "Histogram")
    metrics.set_property("Buckets", [1, 5, 10, 30, 60])
    metrics.set_property("CheckpointSource", "s3://my-bucket/checkpoint.pt")

@metric_scope
def load_checkpoint(model, checkpoint_path: str, metrics: MetricsLogger):
    start_time = time.time()

    # Load model checkpoint
    model.load_state_dict(torch.load(checkpoint_path))

    duration = time.time() - start_time
    log_checkpoint_restore(metrics, duration)

    print(f"Checkpoint restored in {duration} seconds")
```

Prometheus 和 Grafana 範例

此範例示範如何使用 Python 中的 Prometheus 用戶端程式庫來檢測批次應用程式，以長條圖`checkpoint_restore_duration_seconds`的形式公開：

```
from prometheus_client import Histogram
```

```
from prometheus_client import start_http_server
import torch

start_http_server(8080)
restore_duration = Histogram(
    'checkpoint_restore_duration_seconds',
    'Time to restore checkpoint',
    buckets=[1, 5, 10, 30, 60]
)
with restore_duration.time():
    model.load_state_dict(torch.load("s3://my-bucket/checkpoint.pt"))
```

在 Grafana 中，使用查詢`histogram_quantile(0.95, sum(rate(checkpoint_restore_duration_seconds_bucket[5m]) by (le))`將 P95 還原延遲趨勢視覺化。若要進一步了解，請參閱 Prometheus 直方圖文件和 Prometheus 用戶端文件。

應用程式擴展和效能

> **Tip**
>
> 註冊即將舉行的 Amazon EKS AI/ML 研討會。

管理 ML 成品、服務架構和啟動最佳化

在 Amazon EKS 上部署機器學習 (ML) 模型需要考慮模型如何整合到容器映像和執行期環境中。這可確保可擴展性、重現性和高效的資源使用率。本主題說明處理 ML 模型成品、選取服務架構，以及透過預先快取等技術最佳化容器啟動時間的不同方法，所有方法都專為縮短容器啟動時間而量身打造。

在 部署中處理 ML 模型成品

關鍵決策是如何處理 ML 模型成品（例如權重和組態）。選擇會影響影像大小、部署速度、模型更新頻率和操作額外負荷。請注意，當參考儲存「模型」時，我們指的是模型成品（例如訓練過的參數和模型權重）。有不同的方法來處理 Amazon EKS 上的 ML 模型成品。每個 都有其權衡，最佳權衡取決於模型的大小、更新節奏和基礎設施需求。從最低到最高建議考慮以下方法：

- 將模型製作到容器映像中：在映像建置期間，將模型檔案（例如 .safetensors、.pth、.h5) 複製到容器映像（例如 Dockerfile)。模型是不可變影像的一部分。對於不常更新的小型模型，我們建議使用此方法。這可確保一致性和可重現性、提供無載入延遲，並簡化相依性管理，但會導致更大的映像大

小、降低建置和推送速度、需要重建和重新部署模型更新，而且由於登錄提取輸送量，它不適用於大型模型。

- 在執行時間下載模型：在容器啟動時，應用程式會透過啟動容器或進入點中的指令碼，從外部儲存體（例如，由 S3 CRT 支援的 Amazon S3，以使用 S3 CSI 驅動程式掛載點、AWS S3 CLI 或 s5cmd OSS CLI 等方法進行最佳化的高輸送量傳輸）下載模型。對於經常更新的大型模型，我們建議您從此方法開始。這可讓容器映像專注於程式碼/執行時間、無需重建即可輕鬆進行模型更新、支援透過儲存中繼資料進行版本控制，但會導致潛在的網路故障（需要重試邏輯）、需要身分驗證和快取。

若要進一步了解，請參閱《AI on EKS 研討會》中的加速提取程序。

提供 ML 模型

在 Amazon EKS 上部署和提供機器學習 (ML) 模型需要選擇適當的模型服務方法，以最佳化延遲、輸送量、可擴展性和操作簡單性。選擇取決於您的模型類型（例如語言、視覺模型）、工作負載需求（例如即時推論）和團隊專業知識。常見的方法包括用於原型建構的 Python 型設定、用於生產級功能的專用模型伺服器，以及用於高效能和效率的專用推論引擎。每個方法都涉及設定複雜性、效能和資源使用率的權衡。請注意，服務架構可能會因為相依性而增加容器映像大小（多個 GBs)，這可能會影響啟動時間，請考慮使用成品處理技術進行解耦來緩解這種情況。選項從最低到最高建議列出：

使用 Python 架構（例如 FastAPI、HuggingFace 轉換器與 PyTorch) 在容器化節點設定中，使用 Python 架構、內嵌模型檔案（權重、組態、權杖化工具）開發自訂應用程式。

- Pros：易於原型設計，僅 Python，無需額外的基礎設施，相容於所有 HuggingFace 模型，簡單的 Kubernetes 部署。
- Cons：限制為單一請求/簡單批次處理、緩慢產生字符（沒有最佳化的核心）、記憶體效率低下、缺少擴展/監控，以及涉及較長的啟動時間。
- 建議：用於需要自訂邏輯整合的初始原型或單一節點任務。

使用專用模型服務架構（例如 TensorRT-LLM、TGI) 採用 TensorRT-LLM 或 TGI 等特殊伺服器進行 ML 推論、管理模型載入、路由和最佳化。這些支援安全張量等格式，具有選用的編譯或外掛程式。

- Pros：提供批次處理（靜態/傳輸中或連續）、量化 (INT8、FP8、GPTQ)、硬體最佳化 (NVIDIA、AMD、Intel、Inferentia) 和多 GPU 支援（張量/管道平行處理）。TensorRT-LLM 支援各種模型 (LLMs、Encoder-Decoder)，而 TGI 則利用 HuggingFace 整合。
- Cons：TensorRT-LLM 需要編譯且僅限 NVIDIA；TGI 的批次處理效率可能較低；兩者都會增加組態額外負荷，且可能不適合所有模型類型（例如非轉換器）。

- 建議：適用於需要生產功能的 PyTorch/TensorFlow 模型，例如 A/B 測試或具有相容硬體的高輸送量。

使用專門的高輸送量推論引擎（例如 vLLM) 利用 vLLM 等進階推論引擎、使用 PagedAttention 最佳化 LLM 服務、傳輸中批次處理和量化 (INT8、FP8-KV、AWQ)，可與 EKS Autoscaling 整合。

- 優點：高輸送量和記憶體效率（節省 40-60% VRAM)、動態請求處理、字符串流、單節點 Tensor 平行多 GPU 支援，以及廣泛的硬體相容性。
- Cons：針對僅限解碼器的轉換器（例如 LLaMA) 進行最佳化，對於非轉換器模型效率較低，需要相容的硬體（例如 NVIDIA GPUs) 和設定工作。
- 建議：EKS 上大量、低延遲 LLM 推論的首選，可最大化可擴展性和效能。

最佳化容器映像提取時間

大型容器映像可能會導致冷啟動延遲，進而影響 Pod 啟動延遲。對於延遲敏感工作負載，例如水平擴展的即時推論工作負載，快速 Pod 啟動至關重要。請考慮下列方法來最佳化容器映像提取時間：

減少容器映像大小

在啟動期間減少容器映像的大小是使映像變小的另一種方式。您可以在容器映像建置程序的每個步驟進行減少。若要開始，請選擇包含所需最少數量相依性的基礎映像。在映像建置期間，僅包含必要的基本程式庫和成品。建置映像時，請嘗試結合多個 RUN或 COPY命令來建立較少數量的較大層。對於 AI/ML 架構，請使用多階段建置來分隔建置和執行時間、僅複製必要的成品（例如，透過 COPY –from= 用於登錄檔或本機內容），以及選取變體，例如僅限執行時間的影像（例如，3pytorch/pytorch:2.7.1-cuda11.8-cudnn9-runtime.03 GB 與 6.66 GB 的 Devel)。若要進一步了解，請參閱 EKS 上的 AI 研討會中的減少容器映像大小。

減少影像提取延遲

對於在私有子網路中執行的 EKS 叢集，請設定 Amazon ECR 的 VPC 介面端點，以保持 AWS 私有網路上的影像提取流量，減少延遲並消除 NAT 閘道資料處理費用。如需設定指示，請參閱建立 Amazon ECR 的 VPC 端點。

此外，如果您的工作負載依賴外部登錄檔的上游容器映像，請考慮使用 ECR 提取快取來代理和快取私有 ECR 登錄檔中的這些映像。

最佳化映像提取的網路頻寬

提取大型容器映像時，EKS 工作者節點可用的網路頻寬會直接影響訓練和推論工作負載的time-to-start。所有目前世代的 Nitro 型執行個體都使用彈性網路轉接器 (ENA)，但可用頻寬會因執行個體大小而有很大差異，因此了解基準頻寬和高載頻寬之間的差異至關重要（請參閱 Amazon EC2 執行個體網路頻寬)。若要將工作負載的time-to-ready降至最低，請選取相對於映像大小具有足夠基準網路頻寬的執行個體大小，並提取並行。

使用 SOCI 快照器預先提取影像

對於您無法輕鬆最小化的超大型映像，您可以使用平行提取和解壓縮模式中設定的開放原始碼 Seekable OCI (SOCI) 快照器。此解決方案可讓您使用現有的映像，而無需重建或修改建置管道。此選項在將具有非常大型映像的工作負載部署到高效能 EC2 運算執行個體時特別有效。它適用於高輸送量聯網和高效能儲存組態，如同擴展 AI/ML 工作負載一般。

SOCI 平行提取/解壓縮模式透過可設定的平行化策略改善end-to-end映像提取效能。更快的映像提取和準備會直接影響部署新工作負載和有效率地擴展叢集的速度。影像提取有兩個主要階段：

1。從登錄檔擷取層到節點

對於圖層擷取最佳化，SOCI 會為每個圖層建立多個並行 HTTP 連線，將下載輸送量乘以超過單一連線限制。它會將大型分層分割為區塊，並在多個連線中同時下載它們。此方法有助於飽和您的可用網路頻寬，並大幅縮短下載時間。這對單一 layer 可以是數 GB 的 AI/ML 工作負載來說特別重要。

2. 解壓縮和準備這些層以建立容器

為了進行 layer unpacking 最佳化，SOCI 會同時處理多個 layer。其使用您可用的 CPU 核心同時解壓縮和擷取多個層，而不是等待每個層完全解壓縮，再開始下一個層。此平行處理會將傳統 I/O 繫結的解壓縮階段轉換為 CPU 最佳化操作，以隨您可用的核心進行擴展。系統會仔細協調此平行化，以維持檔案系統一致性，同時最大化輸送量。

SOCI 平行提取模式使用雙閾值控制系統搭配可設定的參數，用於下載並行和解壓縮平行處理。此精細控制可讓您微調 SOCI 的行為，以符合您的特定效能需求和環境條件。了解這些參數可協助您最佳化執行時間，以獲得最佳提取效能。

參考

- 如需解決方案和調校權衡的詳細資訊，請參閱 GitHub 上 SOCI 專案儲存庫中的功能文件。

- 如需在 Amazon EKS 上使用 Karpenter 的實作範例，請參閱使用 SOCI 快照器平行提取/解壓縮模式的 Karpenter 藍圖。
- 如需設定 Bottlerocket 進行平行提取的詳細資訊，請參閱 Bottlerocket 文件中的社交快照平行提取解壓縮模式。o

使用 EBS 快照預先提取映像

您可以拍攝快取容器映像的 Amazon Elastic Block Store (EBS) 快照，並為 EKS 工作者節點重複使用此快照。這可確保在節點啟動時在本機預先擷取映像，從而縮短 Pod 初始化時間。如需針對受管節點群組使用 Karpenter 和 EKS Terraform 藍圖的詳細資訊，請參閱使用 Bottlerocket 資料磁碟區縮短 Amazon EKS 上的容器啟動時間。

若要進一步了解，請參閱 AI on EKS 研討會中的使用容器化快照器和將容器映像預先載入 Bottlerocket 資料磁碟區與 EBS 快照。 https://awslabs.github.io/ai-on-eks/docs/guidance/container-startup-time/accelerate-pull-process/prefecthing-images-on-br

使用容器執行期快取來預先提取映像

您可以使用 Kubernetes 資源 （例如 DaemonSet 或 Deployment) 將容器映像預先提取至節點，以填入節點的容器執行期快取。容器執行期快取是由容器執行期管理的本機儲存體 （例如，從登錄檔提取影像後存放影像的容器。 預先提取可確保映像可在本機使用，避免在 Pod 啟動期間發生下載延遲。 當映像經常變更 （例如頻繁更新）、EBS 快照未預先設定、建置 EBS 磁碟區比直接從容器登錄檔提取更耗時，或節點已在叢集中且需要使用多個可能映像之一隨需啟動 Pod 時，這種方法特別有用。

預先提取所有變體可確保快速啟動時間，無論需要哪個映像。例如，在需要 100，000 個使用 10 種不同技術建置之小型模型的大規模平行 ML 工作負載中，透過大型叢集 （例如數千個節點） 中的 DaemonSet 預先提取 10 個映像，可將 Pod 啟動時間降至最低，避免隨需提取在 10 秒內完成。使用容器執行期快取方法不需要管理 EBS 快照， 會確保您一律使用 DaemonSets 取得最新的容器映像版本，但對於節點擴展/擴展的即時推論工作負載，Cluster Autoscaler 等工具新增的新節點可能會在預先提取 DaemonSet 完成映像提取之前排程工作負載 Pod。這可能會導致新節點上的初始 Pod 觸發提取，進而可能延遲啟動並影響低延遲需求。此外，當磁碟用量超過特定閾值或超過設定的最長未使用存留期時，kubelet 映像廢棄收集可以透過移除未使用的映像來影響預先提取的映像。在向內擴展/向外擴展模式中，這可能會移出閒置節點上的映像，這需要在後續擴展期間重新提取，並降低高載工作負載的快取可靠性。

如需將映像預先提取至容器執行期快取的範例，請參閱 AWS GitHub 儲存庫。

考慮 NVMe for kubelet 和容器儲存

請考慮設定 `kubelet`和 `containerd` 以使用暫時性 NVMe 執行個體儲存磁碟，以提高磁碟效能。容器提取程序涉及從登錄檔下載容器映像，並將其層解壓縮為可用的格式。若要在解壓縮期間最佳化 I/O 操作，您應該評估為容器主機執行個體類型提供更高水準的 I/O 效能和輸送量：具有本機儲存體的 NVMe 後端執行個體與 EBS 磁碟區 IOPS/輸送量。對於具有 NVMe 本機儲存的 EC2 執行個體，請考慮為 kubelet (`/var/lib/kubelet`)、 containerd (`/var/lib/containerd`) 和 Pod 日誌 (`/var/log/pods`) 設定節點的基礎檔案系統，以使用暫時性 NVMe 執行個體儲存磁碟以獲得更高層級的 I/O 效能和輸送量。

節點的暫時性儲存可以在請求暫時性儲存的 Pod 之間共用，以及請求下載到節點的容器映像。
如果將 Karpenter 與 Bottlerocket 或 AL2023 EKS Optimized AMIs 搭配使用，這可以透過將 instanceStorePolicy 設定為 RAID0，或在 NodeConfig 中將 localStoragePolicy 設定為使用者資料的一部分，在 EC2NodeClass 中設定。

本指南的貢獻

任何人都可以對最佳實務指南做出貢獻。EKS 最佳實務指南是以 GitHub 上的 AsciiDoc 格式撰寫。

現有參與者的摘要

- bpg-docs.code-workspace 使用 VS 程式碼開啟 ，以自動安裝 AsciiDoc 延伸模組。
  - 進一步了解 Visual Studio Marketplace 上的 AsciiDoc 延伸模組。
- AWS Docs 網站的來源檔案存放在 latest/bpg
- 語法與 Markdown 非常相似。
  - 檢閱 AsciiDoctor 文件中的語法參考。
- 文件平台只會部署 latest/bpg/images。每個指南區段都有返回此目錄的符號連結。例如，latest/bpg/networking/images 指向 latest/bpg/images。

設定本機編輯環境

如果您打算經常編輯指南，請設定本機編輯環境。

分叉並複製儲存庫

您需要熟悉 git、 github和文字編輯器。如需 git和 入門的資訊github，請參閱 GitHub 文件中的 GitHub 帳戶入門。 GitHub

1. 檢視 GitHub 上的 EKS 最佳實務指南。
2. 建立專案儲存庫的分支。了解如何在 GitHub 文件中授予儲存庫。 GitHub
3. 複製專案儲存庫的分支。了解如何複製您的分叉儲存庫。

開啟 VS 程式碼工作區

AWS 建議使用 Microsoft 的 Visual Studio Code 來編輯指南。如需 VS 程式碼的詳細資訊，請參閱 Visual Studio 程式碼文件中的下載 Visual Studio 程式碼和開始使用 Visual Studio 程式碼。

1. 開啟 VS Code。

2. 從複製的儲存庫開啟 bpg-docs.code-workspace 檔案。
3. 如果這是您第一次開啟此工作區，請接受 提示來安裝 AsciiDoc 延伸模組。此副檔名會檢查 AsciiDoc 檔案的語法，並產生即時預覽。
4. 瀏覽至 latest/bpg目錄。此目錄會保留部署至 AWS 文件網站的來源檔案。來源檔案依指南區段組織，例如「安全性」或「網路」。

編輯檔案

1. 在編輯器中開啟檔案。
   - 檢視 AsciiDoc 語法，了解如何建立標題、連結和清單。
   - 您可以使用 Markdown 語法來格式化文字、建立清單和標題。您無法使用 Markdown 語法建立連結。
2. 開啟頁面的即時預覽。
   - 首先，按 ctrl-k或 cmd-k（取決於鍵盤）。第二，按 v。這會在分割檢視中開啟預覽。

AWS 建議使用功能分支來組織您的變更。了解如何使用 git 建立分支。

提交提取請求

您可以從 GitHub 網站或 GitHub cli 建立提取請求。

了解如何使用 GitHub 網站從分支建立提取請求。

了解如何使用 GitHub cli 建立提取請求。

使用 github.dev Web 型編輯器

github.dev Web 型編輯器是以 VS 程式碼為基礎。這是編輯多個檔案和預覽內容的好方法，無需任何設定。

它支援 AsciiDoc 延伸模組。您可以使用 GUI 執行 git 操作。Web 型編輯器沒有執行命令的 shell 或終端機。

您必須擁有 GitHub 帳戶。如有需要，系統會提示您登入。

啟動 GitHub Web 型編輯器。

編輯單一頁面

您可以使用 GitHub 快速更新個別頁面。每個頁面底部都包含「#在 GitHub 上編輯此頁面」連結。

1. 導覽至本指南中您要編輯的頁面
2. 按一下底部的「在 GitHub 上編輯此頁面」連結
3. 按一下 GitHub 檔案檢視器右上角的編輯鉛筆圖示，或按 e
4. 編輯 檔案
5. 使用「遞交變更...」按鈕提交您的變更。此按鈕會建立 GitHub 提取請求。指南維護者將檢閱此提取請求。檢閱者將核准提取請求或請求變更。

檢視並設定頁面的 ID

此頁面說明如何檢視和設定頁面 ID。

頁面 ID 是可識別文件網站上每個頁面的唯一字串。當您在特定頁面上時，可以在瀏覽器的地址列中檢視頁面 ID。頁面 ID 用於 URL、檔案名稱和 ，以建立交叉參考連結。

例如，如果您正在檢視此頁面，瀏覽器地址列中的 URL 看起來會類似：

```
https://docs.aws.amazon.com/view-set-page-id.html
```

URL (`view-set-page-id`) 的最後一個部分是頁面 ID。

設定頁面 ID

建立新頁面時，您需要在來源檔案中設定頁面 ID。頁面 ID 應該是描述頁面內容的簡潔連字號字串。

1. 在文字編輯器中開啟新頁面的來源檔案。
2. 在檔案頂端，新增以下行。它應該位於第一個標題上方。

   ```
   [#my-new-page]
   ```

   `my-new-page` 將 取代為新頁面的頁面 ID。
3. 儲存檔案。

> **Note**
>
> 頁面 IDs 在整個文件網站上必須是唯一的。如果您嘗試使用現有的頁面 ID，您會收到建置錯誤。

建立新頁面

了解如何建立新頁面並更新 內容的指南表格。

建立頁面中繼資料

1. 決定頁面標題和頁面簡短標題。頁面簡短標題是選用的，但如果頁面標題超過幾個字，則建議使用。
2. 判斷頁面的 ID。這在 EKS 最佳實務指南中必須是唯一的。慣例是使用所有小寫，並將單字與 分開-。
3. 視需要在資料夾中建立新的 asciidoc 檔案，並將下列文字新增至檔案：

   Example

   【."topic"】 【#<page-id>】 = <page-title> ：info_titleabbrev： <page-short-title>

   例如

   Example

   【."topic"】 【#scalability】 = EKS 可擴展性最佳實務：info_titleabbrev：可擴展性

新增至目錄

1. 開啟 目錄中父頁面的檔案。對於新的頂層指南區段，父檔案為 book.adoc。
2. 在父檔案底部，更新並插入下列指令：

   Example

   包括：：<new-filename>【leveloffset=+1】

   例如，

Example

包括：：dataplane.adoc【leveloffset=+1】

插入映像

1. 尋找您正在編輯之頁面的影像字首。檢閱 檔案標題中的 `:imagesdir:` 屬性。如需範例，`:imagesdir: images/reliability/`
2. 將您的映像放在此路徑中，例如 `latest/bpg/images/reliability`
3. 為您的映像決定適當的 alt-text。撰寫影像的簡短高階描述。例如，「具有三個可用區域的 VPC 圖表」是適當的 alt-text。
4. 使用 alt-text 和映像檔案名稱更新下列範例。在所需的位置插入 。

   Example

   image：：<image-filename>【<image-alt-text>】

   例如

   Example

   image：：eks-data-plane-connectivity.jpeg【網路圖表】

使用 Vale 檢查樣式

1. 安裝 Vale CLI。
2. 執行 `vale sync`
3. 從 Visual Studio Marketplace 安裝 Vale 延伸模組。
4. 重新啟動 VS 程式碼，並開啟 AsciiDoc 檔案
5. VS 程式碼會強調有問題的文字。了解如何在 VS Code 文件中使用錯誤和警告。

建置本機預覽

1. 在 Linux 或 MacOS `brew` 上使用 安裝`asciidoctor`工具
   - 了解如何在 AsciiDoctor 文件中安裝 Asciidoctor cli。 AsciiDoctor

- 了解如何安裝 brew 套件管理員。

2. 開啟終端機，然後導覽至 `latest/bpg/`
3. 執行 `asciidoctor book.adoc`
   - 檢閱任何語法警告和錯誤
4. 開啟`book.html`輸出檔案。
   - 在 MacOS 上，您可以執行 `open book.html` 在預設瀏覽器中開啟預覽。

AsciiDoc Cheat Sheet

基本格式化

```
*bold text*
_italic text_
`monospace text`
```

標頭

```
= Document Title (Header 1)
== Header 2
=== Header 3
==== Header 4
===== Header 5
====== Header 6
```

清單

未排序的清單：

```
- Item 1
- Item 2
-- Subitem 2.1
-- Subitem 2.2
- Item 3
```

排序清單：

```
. First item
```

```
. Second item
.. Subitem 2.1
.. Subitem 2.2
. Third item
```

連結

```
External link:  https://example.com[Link text]
Internal link: <<page-id>>
Internal link: xref:page-id[Link text]
```

映像

```
image::image-file.jpg[Alt text]
```

程式碼區塊

```
 [source,python]
 ----
 def hello_world():
     print("Hello, World!")
 ----
```

表格

了解如何建置基本資料表。

```
[cols="1,1"]
|===
|Cell in column 1, row 1
|Cell in column 2, row 1

|Cell in column 1, row 2
|Cell in column 2, row 2

|Cell in column 1, row 3
|Cell in column 2, row 3
|===
```

輔助

```
NOTE: This is a note admonition.

WARNING: This is a warning admonition.

TIP: This is a tip admonition.

IMPORTANT: This is an important admonition.

CAUTION: This is a caution admonition.
```

預覽版：

> **Note**
>
> 這是筆記提醒。

包括

```
include::filename.adoc[]
```

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。